A CUDA-enabled Hadoop Cluster for Fast Distributed Image Processing

Ranajoy Malakar
Corporate Research and Technology
Siemens Technology Services
Bangalore, India
ranajoy.malakar@siemens.com

Naga Vydyanathan
Corporate Research and Technology
Siemens Technology Services
Bangalore, India
vydyanathan.nagavijaylakshmi@siemens.com

***Abstract*—Hadoop is a map-reduce based distributed processing framework, frequently used in the industry today, in areas of big data analysis, particularly text analysis. Graphics processing units (GPUs), on the other hand, are massively parallel platforms with attractive performance to price and power ratios, used extensively in the recent years for acceleration of data parallel computations. CUDA or Compute Unified Device Architecture is a C-based programming model proposed by NVIDIA for leveraging the parallel computing capabilities of the GPU for general purpose computations. This paper attempts to integrate CUDA acceleration into the Hadoop distributed processing framework to create a heterogeneous high performance image processing system. As Hadoop primarily is used for text analysis, this involves facilitating efficient image processing in Hadoop. Our experimental evaluations using a Adaboost based face detection algorithm indicate that CUDA-enabling a Hadoop cluster, even with low-end GPUs, can result in a 25% improvement in data processing throughput, indicating that an integration of these two technologies can help build scalable, high throughput, power and cost-efficient computing platforms.**

***Index Terms*—Hadoop; Map-reduce; CUDA; GPGPU.**

INTRODUCTION

The Hadoop software library [1] is a highly scalable framework that allows for the distributed processing of large data sets across clusters of computers using a simple programming model. Hadoop based systems are now heavily used by leading players in the industry like Yahoo, Facebook, Amazon etc. Hadoop is more common in text analysis and machine learning domains while processing of image and binary data is comparatively less explored.

GPGPU stands for general-purpose computation on graphics processing units (GPUs). GPUs are massively parallel many-core processors that can be used to accelerate a wide range of data-parallel general purpose applications like image processing and analysis, in addition to graphics processing. Several works have demonstrated the power of GPUs in providing orders of magnitude computational speedups in the image processing domain [2], [3], [4]. CUDA [5] is a programming abstraction from NVIDIA that facilitates general purpose computations on NVIDIA GPUs.

While Hadoop offers speed and scalability by distribution of the data and computation over multiple nodes, CUDA offers acceleration of the computation on a single node by utilizing the massively parallel processing cores of the graphics cards. In this paper, we explore the possibilities of integrating CUDA acceleration into the Hadoop framework to build a heterogeneous high performance image processing system. We evaluate such a system in the context of a face processing algorithm based on the Adaboost learning system that detects human faces in image streams [3]. Our experimental evaluations indicate that CUDA-enabling a small Hadoop cluster, even with low-cost, low-end GPUs, can result in a 25% improvement in processing throughput, thereby indicating that an amalgamation of these two technologies can lay the foundation for building scalable, high throughput, low-cost and efficient computing platforms.

BACKGROUND

Hadoop

Hadoop [1] is a popular software framework used for data intensive distributed processing in the area of big data analytics. Internally, Hadoop is a Java implementation of MapReduce [6], which is a popular software architecture that facilitates processing of large amounts of data in a distributed fashion. Hadoop provides a collection of Java classes, centered on Mapper and Reducer interfaces (Figure 1). The Mapper typically maps the input data to a list of key-value pairs and the reducer combines the values of a particular key to produce the desired final output. The application needs to create its own Mapper and Reducer implementations, register the Mapper and Reducer classes into a Hadoop job, indicate the location of the input and output, and fire it to the Hadoop framework. The framework takes care of reading the data from the input location, invokes the Mapper and Reducer application classes when needed in a concurrent and distributed fashion, and writes the result to the output location. Hadoop input and output are always read from the Hadoop Distributed File System (HDFS).

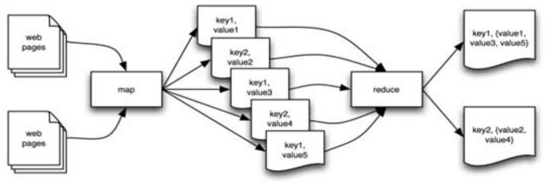

Fig 1: Hadoop programming model (Courtesy adcalves.wordpress.com [7]).

There are multiple flavors of Hadoop available in the market like Apache [1] and Cloudera [8]. Hadoop applications are usually coded in Java.

CUDA

Compute Unified Device Architecture (CUDA) [5] is a C based programming model from NVIDIA that exposes the many-core capabilities of the GPU for easy development and deployment of general purpose computations. In the CUDA context, the GPU is called as a device, whereas the CPU is called as host. A kernel is a set of computations that is offloaded by the CPU to be executed on the GPU. A CUDA kernel is executed on the GPU by a grid of thread blocks, each consisting of a set of threads. For example, the Fermi GPU [9] can support up to 20K concurrent threads while the Kepler GPU [10] can support up to 30K concurrent threads. As the GPU has hundreds of processing cores that are capable of doing efficient data-parallel computations, CUDA allows the programmer to offload compute-intensive data-parallel portions of their applications as kernels, to the GPU. Both industry and academia have reported obtaining orders of magnitude of speedup through CUDA acceleration [2], [3], [4]. CUDA programs are usually written in C/C++, as CUDA provides binding for the C++ language.

Related Work

GPU accelerated Hadoop clusters have been explored in the past [11], [12], [13]. The first work that demonstrated the capability of CUDA acceleration on a single machine running the Phoenix map-reduce framework [14] was by He~et~al. [11] called Mars. Mars was restricted to textual analytics and was not designed to scale across a GPU cluster. Also, additional APIs outside of the original map-reduce framework were introduced in Mars to count the size of the result and to output the result. Farivar~et.~al [12] designed a system called MITHRA that integrated CUDA acceleration of Monte-carlo computations into Hadoop using streams [15]. Hadoop streams allow any executable to be run as the mapper or reducer in the map-reduce framework and the input and output to the executable is passed through stdin and stdout. This approach incurs heavy overhead [16] and is not scalable especially while processing image or binary data. In addition, all of these works do not evaluate a truly heterogeneous system that performs computations simultaneously on both the multi-cores as well as the GPUs. Yet another approach to integrate CUDA into hadoop is through JCUDA [13]. We discuss this approach and Hadoop streams in detail in the next section.

CUDA-Enabled Hadoop for Image Processing

There are broadly three approaches to execute a CUDA function from within a Hadoop map reduce job. This restriction arises due to the fact that CUDA bindings are by default to the C or C++ language, whereas the Hadoop map reduce jobs are usually written in the Java programming language. We first outline each of these approaches in detail and justify the approach adopted by us.

Hadoop Streaming

Hadoop streaming [15] is an utility supplied with Hadoop distributions which allows the user to create and run map-reduce jobs with any executable or script as the mapper and/or the reducer. The executables need to be designed to read their input from stdin and write their output to stdout. Each mapper task launches the specified executable as a separate process during initialization of the mapper. As the mapper task runs, it converts its inputs into lines and feeds the lines to the stdin of the process. In the meantime, the mapper collects the line oriented outputs from the stdout of the process and converts each line into a key/value pair, which is collected as the output of the mapper. This approach has the obvious disadvantage in the fact that external process creations are slow, and therefore, there is a perceptible slowdown (~30%) compared to native java approaches [16]. Also, this approach presents a problem when dealing with image and binary data, as the data must be formatted to be suitable for input/output to stdin/stdout for consumption of the Java based mapper/reducers.

JCUDA

JCUDA [17] is a Java binding for the NVIDIA CUDA library. JCUDA involves writing CUDA programs as strings in the Java program, which then are compiled just in time before the CUDA code executions. While there are possibilities to bypass this approach, most JCUDA programs just use the JIT method as the performance loss due to the occurrence of the just in time compilation is usually amortized by sufficiently large input data sets being processed by the CUDA code.

JCUDA provides a higher level of abstraction over calling CUDA kernels through Java Native Interface (JNI) and hence offers better programmability at the cost of larger overheads as compared to JNI. Since, in the Hadoop framework, every GPU-mapper instance would invoke the CUDA kernels, we decided to use JNI rather than JCUDA, to minimize the recurring overheads. In addition, JCUDA imposes certain restrictions on the type of primitives that can be passed to the

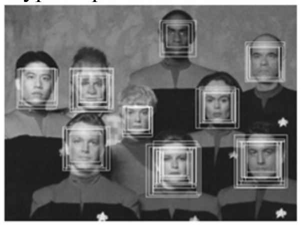

Fig 2: Sample output image of the Adaboost-based face detection algorithm

CUDA kernels and does not provide direct support for re-use of data across kernels [17].

Java native interface

The Java Native Interface (JNI) is a programming framework that enables Java code running in a Java Virtual Machine (JVM) to call native applications (programs specific to a hardware and operating system platform) and libraries written in other languages such as C, C++ and assembly. Since CUDA code is written in C++, and Hadoop map reduce jobs in Java, JNI can be used to invoke the CUDA kernels from the map-reduce jobs. JNI is efficient with minimal overheads, and is widely used and accepted in the industry. We chose JNI for its simplicity of usage and low overheads.

We evaluated the performance impact of CUDA acceleration of Hadoop using an Adaboost-based face detection algorithm presented by Sharma~et.~al. [3]. The face detection algorithm works on one image at a time, and attempts to detect the existence of faces in the image. Any detected faces are marked with a white box, and the resultant image is created as output. Figure 2 shows a sample output image of the face detection algorithm.

The face detection algorithm was implemented in multiple versions - a sequential version in C++ that ran on a single core and a CUDA-based version that uses GPUs for acceleration. We recorded video in Siemens office premises, and split the video into frames to form our source input images. This created a very real world scenario where zero faces in an image to multiple faces, even crowds were captured as input image. Since each image frame was quite small, i.e about 76 KB in size, we had to ensure the efficient handling of small files in Hadoop.

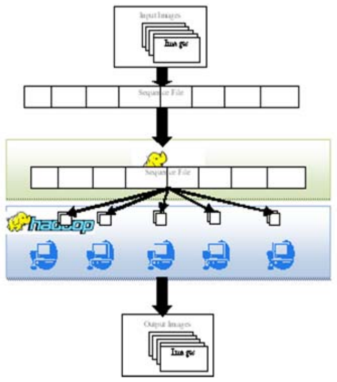

Fig 3: Image processing in Hadoop using sequence files

Handling small files in Hadoop

In video stream processing, every frame is typically much smaller in size than the Hadoop Distributed File System (HDFS) block size, which is 64 MB by default. For example, in our face detection example, each image frame was about 76 KB.

Every file, directory and block in HDFS is represented as an object in the HDFS's namenode and each of these objects in the namenode occupies around 150 bytes. Therefore having a large numbers of small files results in a lot of spatial overhead in HDFS. Furthermore, HDFS, which is primarily designed for streaming access of large files, is not tuned for efficient access of small files. Reading through small files normally causes lots of seeks and hopping from datanode to datanode, and hence is inefficient [18]. In addition, if every map task processes very little data, the overheads of launching and managing a large number of map tasks becomes substantial in comparison to the overall processing time.

In order to circumvent this problem, we used Hadoop sequence files. Hadoop sequence files are flat files consisting of binary key and value pairs. In our design, we assigned the names of the images as the keys and the binary data of the images as the values in our sequence file. The sequence file, thus, is a collection of images and hence is large enough for efficient storage and retrieval in HDFS. Sequence files have "sync marker"s introduced into the format, in order to ensure that a record is read in its entirety, even when the sequence file is split in blocks across nodes in HDFS. For a large sequence file that is split across nodes, a mapper that works on the first block does not finish reading at the split boundary, but rather continues reading till the next sync marker location. Subsequent mapper tasks reading from the second block ensure that they begin reading from the sync marker location and ignore content before that [19]. Simply put, Hadoop automatically ensures that records are never read partially by the mapper, and thus in our case, no split image is supplied for analysis. Even though this automatic splitting using Hadoop is modifiable, for our case it was not required, and we retained the above behavior for our experiments. Also, the size of each record in our case was far smaller than the default HDFS split size of 64 MB, which implied that reading beyond split boundaries was not essential for most map tasks in our experiments. Thus, for our Map reduce job, the image sequence file is the input data, and chunks of the sequence file is the input for each of the map tasks at each of the slave nodes.

Hadoop Distributed Cache

Before the sequential or the CUDA-based face detection algorithm can be invoked through JNI, the dynamic modules containing the compiled C++ and CUDA code as well as any associated CUDA runtime modules must be loaded, so that the Java Virtual Machine (JVM) on which the map task is running can locate the necessary modules. Further, the face detection algorithm required some training data files during execution. To facilitate the distribution of the training data as well as the relevant dynamic modules to each node in the cluster where the map function was executed, we used the Hadoop distributed cache.

The Hadoop Distributed Cache enables efficient caching of large read-only files to each slave node in a Hadoop cluster. Applications specify the files to be cached via urls (hdfs://) in the Java code. The distributed cache framework copies the necessary files to a slave node before any tasks in the job are executed on that node, only once per job.

In order to set the environment variables necessary for the JVM to find and load the associated CUDA runtime modules, we used Java APIs from within the ``configure" function in Hadoop. The configure function is invoked before the map functions, and is invoked only once per instance of JVM that Hadoop creates for launching of map functions on slave nodes.

Finally, to summarize, in our distributed face detection application, the map stage applies the Adaboost-based face detection algorithm to each frame of the video stream, while the reduce stage is just the identity reducer provided by Hadoop. Both a sequential and a CUDA-based implementation of the face detection algorithm is made available and the map function invokes either the sequential or the CUDA-based implementation through JNI. The decision of whether the CPU or the GPU should be used for execution of the face detection algorithm is taken, based on probing of the resource usage on the node where the map function is invoked. More details on the probing mechanism are given in the next section.

EXPERIMENTAL RESULTS

Our test setup was a 5 node Hadoop cluster running Cloudera Hadoop [8]. Each node in the cluster was equipped with a Intel Core i7-860 Processor 2.80-GHz, 8MB total cache quad-core processor with hyper-threading enabled, along with 8GB PC3-10600 Memory at 1333MHz. Each node was also equipped with a low-end NVIDIA GeForce 310 graphics processing unit with 16 CUDA capable cores and 512MB of memory, where each core is clocked at1402MHz. Performing these experiments on a larger cluster with more powerful GPUs would have given us more insight and better performance, but unfortunately, such infrastructure was not available with us.

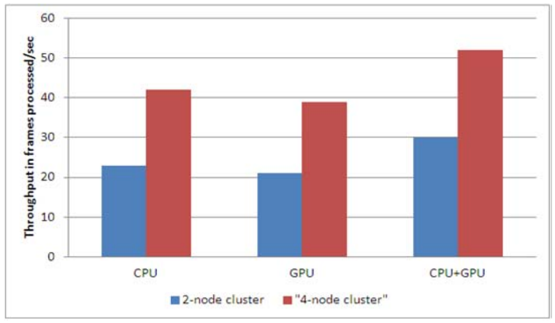

Fig 4: Throughput of face detection on a CUDA accelerated Hadoop cluster.

As mentioned in Section IV each invocation of the map function either calls the sequential CPU based implementation or the CUDA based implementation of the face detection algorithm, depending on the resource usage. For example, for maximum throughput, the Hadoop framework should be able to invoke 9 concurrent map functions on each node, two on each core (4X2), since hyper-threading is enabled and 1 instance on the GPU. The decision as to whether the face detection algorithm is executed on the CPU or GPU is taken within the map function. For this purpose we required an API from CUDA that would enable us to query at any instant if a GPU is busy. Such a querying mechanism however, is unavailable from the GPU vendor at the moment, and therefore, the detection of the GPU's availability for scheduling needs was satisfied in round about fashion. NVIDIA GPUs support a mode of execution called "compute exclusive" mode, in which only a single thread can access the GPU at a time. GPUs can be set to this mode using a command line tool called NVIDIA system management interface (nvidia-smi). Using the nvidia-smi, the GPUs at each node in the cluster was set to compute exclusive mode, and in each map task they were probed for availability. On nodes where GPUs were unavailable, the map task was scheduled on the CPU. To quantify the benefit of integrating CUDA acceleration, we performed three sets of experiments, where the face detection routines are executed on: a) the CPU, b) the GPU and c) both the CPU and the GPU, one each node. Total size of input data was 3.2 gigabytes, with size of individual frames at 76 kilobytes. Total number of frames given as input was around 42000. To evaluate the scalability of the system, we ran our experiments on a 2 node and 4 node subset of the cluster. Figure 4 plots the throughput achieved on a 2 node and a 4 node cluster when only the CPU is used, when only the GPU is used and when both are used simultaneously. As the GPU on each node in the cluster is a low-end GPU with only 16 cores at 1402MHz, while the CPU is a more powerful i7 2.80GHz processor, we notice a dip in the throughput achieved when using only the GPU as compared to using only the CPU. When CUDA acceleration is leveraged in conjunction with the multiple cores, we see the best achieved throughput. In the experiments, as we move from a 2 node to a 4 node cluster, we see a 1.73x to 1.86x scaling in the throughput, thereby showing a good scalability of the Hadoop framework.

The distribution of Hadoop tasks is controlled by the Hadoop scheduler, and the default Hadoop scheduler failed to schedule tasks evenly across nodes in the cluster. It was noted that load distribution at any point of time ended up being skewed towards a few nodes. Also, it was noted that the actual computation took about 55% of processor seconds while for 45% of processor seconds the nodes where either idle or executing Hadoop related management tasks. Consequently, the best performance achievable, which would have been an additive combination of speeds achieved by the GPU and CPU, was not possible. Hadoop does present the possibility of tweaking the scheduler for user needs and also of using a user specific scheduler, but due to time constraints, it was not possible to address these possibilities in the experiments.

CONCLUSION

A CUDA-accelerated Hadoop framework holds a lot of promise as a platform for fast, distributed, computationally intensive processing of large data sets. In this paper, we demonstrate how CUDA can be integrated into the Hadoop framework for creating a high performance distributed image processing system. We also discuss how small image and binary files can be handled efficiently in Hadoop. Our experimental evaluations using an Adaboost based face detection algorithm indicate that CUDA enabling a Hadoop cluster can yield a 25% improvement in throughput, even when using low-end graphics cards. Thus, an integration of CUDA with Hadoop can help build very powerful, distributed computational platforms that are also cost and power efficient from small clusters equipped with mid-range graphics cards.

REFERENCES

“Apache Hadoop,” http://en.wikipedia.org/wiki/Apache/_Hadoop.

Z. Yang, Y. Zhu, and Y. Pu, “Parallel image processing based on cuda,” in Proceedings of the 2008 International Conference on Computer Science and Software Engineering - Volume 03, ser. CSSE ’08. IEEE Computer Society, 2008, pp. 198–201.

B. Sharma, R. Thota, N. Vydyanathan, and A. A. Kale, “Towards a robust, real-time face processing system using cuda-enabled gpus,” in 16th International Conference on High Performance Computing, HiPC, Dec 2009, pp. 368–377.

Y. Okitsu, F. Ino, and K. Hagihara, “High-performance cone beam reconstruction using cuda compatible gpus,” Parallel Comput., vol. 36, no. 2-3, pp. 129–141, Feb. 2010.

“Nvidia cuda c programming guide,” http://developer.download.nvidia.com/compute/cuda/32/toolkit/docs/CUDAn Cn Programmingn Guide.pdf.

J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” Commun. ACM, vol. 51, no. 1, pp. 107–113, Jan. 2008.

"Hadoop programming model,” http://adcalves.wordpress.com/2010/12/12/a-hadoop-primer/

“Cloudera hadoop,” http://www.cloudera.com/content/cloudera/en/why-cloudera/hadoop-and-big-data.html.

“Nvidia fermi architecture for high performance computing,” http://www.nvidia.com/object/fermi-architecture.html.

“Nvidia kepler compute architecture for high performance computing,” http://www.nvidia.com/object/nvidia-kepler.html.

B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, “Mars: amapreduce framework on graphics processors,” in Proceedings of the 17th international conference on Parallel architectures and compilation techniques, ser. PACT ’08, 2008, pp. 260–269

R. Farivar, A. Verma, E. Chan, and R. H. Campbell, “Mithra: Multiple data independent tasks on a heterogeneous resource architecture.” in IEEE International Conference on Cluster Computing, 2009, pp. 1–10.

“Cuda on hadoop,” http://wiki.apache.org/hadoop/CUDAn%20Onn%20Hadoop.

C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis,“Evaluating mapreduce for multi-core and multiprocessor systems,” in Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, ser. HPCA ’07, 2007, pp. 13–24.

“Hadoop streaming,” http://hadoop.apache.org/docs/r0.15.2/streaming.html.

“Performance hadoop stream map reduce,” http://code.google.com/p/hadoop-stream-mapreduce/wiki/Performance.

Y. Yan, M. Grossman, and V. Sarkar, “Jcuda: A programmer-friendly interface for accelerating java programs with cuda,” in Proceedings of the 15th International Euro-Par Conference on Parallel Processing, ser.Euro-Par ’09, 2009, pp. 887–899.

“Hadoop - the small files problem,” http://blog.cloudera.com/blog/2009/02/the-small-files-problem/.

“Splitting sequencefile in controlled manner - hadoop,” http://stackoverflow.com/questions/8405671/splitting-sequencefile-in-controlled-manner-hadoop/.