Adaptively Accelerating Map-Reduce/Spark with GPUs: A Case Study

K. R. Jayaram
IBM Research
Yorktown Heights, NY, USA
jayaramkr@us.ibm.com

Anshul Gandhi
Stony Brook University
Stony Brook, NY, USA
anshul@cs.stonybrook.edu

Hongyi Xin
Carnegie Mellon University
Pittsburgh, PA, USA
hxin@cs.cmu.edu

Shu Tao
IBM Research
Yorktown Heights, NY, USA
shutao@us.ibm.com

***Abstract*—In this paper, we propose and evaluate a simple mechanism to accelerate iterative machine learning algorithms implemented in Hadoop map-reduce (stock), and Apache Spark. In particular, we describe a technique that enables data parallel tasks in map-reduce and Spark to be dynamically and adaptively scheduled on CPU or GPU, based on availability and load. We examine the extent of performance improvements, and correlate them to various parameters of the algorithms studied. We focus on end-to-end performance impact, including overheads associated with transferring data into and out of the GPU, and conversion between data representations in the JVM and on GPU. We also present three optimizations that, in our analysis, can be generalized across many iterative machine learning applications. We present a case study where we accelerate four iterative machine learning applications – multinomial logistic regression, multiple linear regression, K-Means clustering and principal components analysis using singular value decomposition, implemented in three data analytics frameworks – Hadoop Map-Reduce (HMR), IBM Main-Memory Map-Reduce (M3R) and Spark. We observe that the use of GPGPUs decreases the execution time of these applications on HMR by up to 8×, M3R by up to 18× and Spark by up to 25×. Through our empirical analysis, we offer several insights that can be helpful in designing middleware and cluster managers to accelerate map-reduce and Spark applications using GPUs.**

I. Introduction

The popularity and affordability of GPUs has renewed interest in using GPUs for "ordinary" Machine Learning (ML) applications, such as regression, classification, and clustering, beyond the well established use of GPUs for deep learning workloads. Previous research to exploit GPUs for accelerating ML applications [2], [26] typically (i) used traditional C/C++ with CUDA and MPI, with *explicit* co-ordination, failure handling, and distribution of tasks across nodes and between CPU and GPU, or (ii) proposed entirely new GPU-optimized programming frameworks and libraries. Unfortunately, both approaches have limited applicability in the industry due to the widespread popularity and market penetration of established analytics frameworks such as Map-Reduce and Spark, and libraries that build on them (e.g., MLib [9], MLbase [5], Mahout [16]). Since these existing frameworks are routinely employed by ML software in production, they have the advantage of being extensively tested, resulting in lower incidence of erroneous code (when compared to a C++/MPI application written from scratch). Further, the Map-Reduce and Spark frameworks and their commercial offerings automatically handle task distribution, high-availability, fault-tolerance, and elasticity (e.g., Amazon Elastic Map-Reduce [3] and Databricks Elastic Spark [22]), allowing practitioners who use such frameworks to focus on the results of the analyses.

Organizations, including Databricks, which commercializes Spark, have acknowledged the potential of GPUs to significantly improve the performance of ML applications [22]. Analytics frameworks like Apache Hadoop Map-Reduce and Apache Spark, do not automatically generate code that leverages GPUs, but do support the use of *external* GPU-optimized libraries within user code (e.g., within the user's *map* function). Several researchers have attempted to first quantify the performance benefits of using GPUs to accelerate map-reduce [12]. For a detailed discussion, we refer the reader to Section VI. However, despite their claimed performance benefits, these approaches suffer from several drawbacks that have limited their adoption in practice:

- The *end-to-end* performance benefits of using GPUs to accelerate map-reduce has remained unclear. Previous papers have not considered the overhead of transferring data from main memory to GPU memory, network communication overhead, shuffle overhead, and serialization of data between the Java Virtual Machine (JVM) and the GPU runtime.
- Several previous projects have either omitted fault-tolerance from GPU accelerated map-reduce [12], [20], [29], [32], or redesigned the map-reduce API [29], [32], or assumed that all data fits in GPU memory [12] – all of which limit their usefulness in practice.
- Previous systems and papers do not provide evidence of their ability to handle large datasets and have not been evaluated on datasets larger than 1GB.

In this paper, we examine the extent to which the use of GPUs improves the performance of iterative ML algorithms implemented in Apache Hadoop Map-Reduce (HMR) [6] and Apache Spark [18], along with IBM's own Main-Memory Map-Reduce (M3R) framework [31]. We propose a simple mechanism to adaptively distribute data-parallel tasks in said frameworks between CPUs and GPUs. We empirically analyze the performance of four popular iterative ML algorithms on the above frameworks under different parameter settings and different deployment sizes (number of nodes); we employ Amazon EC2 GPU-equipped instances for all our experiments.

Notably, we focus on end-to-end performance benefits (in terms of execution time), and *do not modify* these frameworks. All the code necessary to use GPGPUs is within the application code, e.g., in the case of Hadoop map-reduce, it is within the map and reduce functions. To the best of our knowledge, this is the first paper to accelerate stock implementations of Hadoop Map-Reduce, M3R and Apache Spark on a multi-node cluster combining CPUs and GPUs, without modifying (and affecting) the interfaces, implementations and fault-tolerance properties of these frameworks.

Our analysis provides several interesting insights:

- We demonstrate that speedups are significant (as much as 16–24×), even for highly optimized in-memory analytics platforms such as M3R (1.26–18.46×) and Spark (1.87–24.68×).
- We find that the speedup achieved by using GPUs initially increases with the number of nodes and eventually flatlines due to communication overheads.
- We find that the speedup achieved using GPUs increases with the amount of computation involved in the algorithms.
- We explore several optimizations when implementing iterative algorithms in all three frameworks, and we outline three generic optimizations that provide significant performance improvements.

The rest of the paper is organized as follows. We describe the data analytics frameworks and ML applications we employ in Sections II and III, respectively. We then present our approach to accelerating ML applications, including the optimizations we leverage, in Section IV. We present our evaluation results using four different ML applications on three different data analytics frameworks in Section V. We discuss related work in Section VI and conclude in Section VII.

II. Frameworks

Our goal in this paper is to analyze the performance of popular ML algorithms on commonly employed analytics frameworks when leveraging GPUs. We thus consider the following frameworks and applications in our study.

Hadoop Map-Reduce (HMR): Hadoop Map-Reduce [6] started as the open-source implementation of MapReduce [19], and has since evolved to assimilate insights and optimizations from the thousands of organizations that use it. We use Hadoop Map Reduce [6] version 3.0.3, which also includes the Apache Hadoop YARN [8] cluster manager and the HDFS distributed file system. For the experiments in this paper, no other application frameworks are executing on top of YARN, and the map reduce framework has full access to the GPUs in the cluster, along with all the CPUs and RAM.

Main-Memory Map-Reduce (M3R) [31]: HMR applications involve significant disk access (HDFS) and shuffle overheads; this limits the speedup that GPUs can provide for HMR. To mitigate these overheads, IBM Research developed a new implementation of the HMR API, called Main-Memory Map-Reduce (M3R) [31], targeted at data analytics on high mean-time-to-failure clusters. M3R is implemented in X10 [14], a type-safe, object-oriented, multi-threaded, multi-node, garbage-collected programming language. M3R stores key value (KV) sequences in a family of long-lived JVMs, sharing heap-state between jobs. On input from the file system, M3R associates the input splits with the global (multi-place) KV sequence obtained from this input. Subsequent invocations of the input splits (e.g. by subsequent jobs in the sequence) are fulfilled by reading the KV sequence from the heap, eliminating the need to read from the file system again, and deserialized. Similarly, output to an output formatter is associated with the global KV sequence so that subsequent input requests can be fulfilled from the KV sequence. The shuffle of KV pairs is done in memory, using X10 inter-process communication; this has the benefit of de-duplication performed by the X10 serialization mechanism.

Spark [18] [33]: We also use Apache Spark in our experiments because of its tremendous increase in popularity, and also because it has eliminated several of the overheads of HMR. Analytics applications are significantly faster (5-25×) when programmed in and executed on Apache Spark [33]. We use native Spark (version 2.3.0), i.e., Spark without any cluster manager like YARN or Mesos [25] or Kubernetes [23]. Spark provides a unified framework to manage big data processing requirements with a variety of data sets that are diverse in nature (text data, graph data, etc.) as well as the source of data (batch vs. real-time streaming data). The fundamental programming abstraction of Spark is called Resilient Distributed Datasets (RDDs), an in-memory logical collection of data partitioned across machines. RDDs can be created by referencing datasets in external storage systems, or by applying coarse-grained transformations (e.g. map, filter, reduce, join) on existing RDDs.

Fairness of Evaluation: The frameworks we have chose have different properties, and were invented at different timeframes for different usage scenarios. We choose them mainly because of their popularity and because all three of them are industrial-grade and used in production deployments. All three of them have varying fault-tolerance guarantees. Also, we are forced to use YARN when deploying Hadoop Map-Reduce because that is the only way to do so (beyond version 1, Hadoop Map-Reduce requires YARN while Spark can execute natively on a cluster). Due to these factors, we do not undertake inter-framework performance comparisons in this paper. We focus, instead, on comparing the performance of an application implemented in a framework with and without GPUs.

III. Applications Used

In terms of ML applications, we focus on the broad categories of clustering, classification, regression, and dimensionality reduction; this categorization is based on the output of the algorithm, as suggested by Bishop [13]. We choose one popular algorithm from each class to accelerate using GPUs, as discussed below.

Clustering : K-Means++. K-Means clustering is a method of classifying/grouping N items into K groups/clusters (where K is the number of pre-chosen groups). The grouping is done

by minimizing the sum of squared distances (Euclidean distances) between items and the corresponding cluster centroid. Although finding an exact solution to the K-Means problem for arbitrary input is NP-hard, the standard approach to finding an approximate solution (often called Lloyd's algorithm or the K-Means algorithm) is used widely and frequently finds reasonable solutions quickly. This algorithm's guarantees can be further improved by using a good initialization technique to pick the initial set of centroids – a technique proposed by Arthur and Vassilvitskii called K-Means++ [10], which we use for our case study (as is the case in Spark MLlib and Apache Mahout).

Classification : Multinomial Logistic Regression. Multinomial logistic regression is a classification method that generalizes logistic regression to multiclass problems (more than two classes). It is a model that is used to predict the probabilities of the different possible outcomes of a categorically distributed dependent variable, given a set of independent variables (which may be real-valued, categorical-valued, etc.). Our implementation uses the limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) [13] algorithm, similar to Spark's MLib library [9] for parameter estimation.

Multiple Linear Regression. Multiple Linear regression is an approach for modeling the relationship between a scalar dependent variable y and more than one explanatory variables (or independent variables), denoted as $\vec{X}$. Our implementation uses the linear least squares algorithm with Stochastic Gradient Descent (SGD), similar to Spark's MLib library [9].

Dimensionality Reduction : Principal Components Analysis (PCA) through Singular Value Decomposition (SVD). Dimensionality reduction is the process of reducing the number of random variables under consideration. PCA is a popular dimensionality reduction algorithm, commonly employed in face recognition and image compression, to identify patterns in data and express the data in a lower dimensional space PCA performs a linear mapping of the data to a lower-dimensional space in such a way that the variance of the data in the low-dimensional representation is minimized. In practice, the correlation matrix of the data is constructed and the eigenvectors on this matrix are computed through SVD. SVD, by itself, is one of the cornerstones of linear algebra and has widespread application in signal processing and pattern recognition [13].

IV. Approach to Acceleration

We now describe how we accelerate the ML applications listed above. We start with an overview of our implementation requirements and basic approach. We then discuss the various challenges in effectively accelerating iterative ML applications and the optimizations we employ to address them (Sections IV-C–IV-E).

A. Implementation Constraints

Recall that we aim to accelerate ML applications implemented in *stock* HMR, M3R and Spark without changing the APIs of these frameworks or without redesigning them. This also implies that we perform all GPU offloading inside high-level functions like map(...), combine(...) and reduce(...). This approach aligns with our long-term goals for designing an automated framework, consisting of a compiler and runtime, to transparently accelerate map-reduce and Spark applications. We evaluate all our accelerated applications on realistic platforms.

B. Basic Approach

HMR and M3R require applications to be programmed in Java, while Spark supports Scala, Python and Java. To ensure uniformity and fairness, we choose Java to implement all our applications, and accelerate them by using JCUDA [21]. Consider the map-reduce programming paradigm. For example, to accelerate map using GPUs, we re-implement the logic of map as a CUDA kernel [27], and call it from map using JCUDA bindings. This means that the map task starts executing on a CPU, loads data from HDFS into main memory, and moves this data into GPU memory before launching the CUDA map kernel. This CUDA map kernel internally creates multiple threads (we use 1024 threads), which are then divided by the GPU runtime into thread blocks and warps.

GPUs are effective at massive parallelization; one recommended way to use them is to move enough data into GPU memory, and parallelize computation on said data, while minimizing the frequency of data movement into/out of GPU memory. We thus change the default HDFS block size from 64MB to 128MB. After the kernel finishes processing, control transfers back to the map task on CPU, which returns to the Hadoop scheduler. The map task performing the GPU offloading does not block the CPU because both HMR and M3R already schedule multiple map and reduce tasks on every machine for concurrency. For maximum efficiency, we implement map-reduce combiners [7] in all our applications to perform a local reduce before shuffling and a distributed reduce. We also accelerate combine and reduce using GPUs following a similar approach.

For Spark, although the engine is implemented in Scala and there are no viable native-Scala bindings for CUDA, Spark and the Scala runtime are compatible with Java, and Spark provides a Java API. We implement our Spark applications in Java, and use JCUDA inside the high-level user-defined functions like map, reduceByKey, join, filter , etc. Each Spark task processes an RDD block; we set the RDD block size to be the same as the HDFS block size (128 MB).

C. Adaptive CPU-GPU Scheduling and Load Balancing

Utilizing both the CPU and GPU for a single application is challenging by itself, because different parts of the application may be best suited to (one of) CPU or GPU. This is even more challenging with frameworks like map-reduce and Spark because the framework, and not the programmer, schedules the tasks; further, we want our implementations to work with *existing* HMR, M3R and Spark task schedulers. Also, different applications spend varying fractions of their time on map, reduce and the other stages. Hence, deciding which task gets

the GPU, while ensuring that other tasks proceed in parallel on the CPU, requires co-ordination between tasks.

To explain our approach to CPU-GPU scheduling, consider the example of map. For each stage in the application, all three frameworks take a single map function. In the GPU accelerated version of our application, this function is responsible for offloading computation (i.e. the GPU kernel) once it has been scheduled on an available CPU – the GPU kernel in turn creates several threads (and warps). Each GPU can effectively execute a small number (typically less than 16) of kernels concurrently – we denote this by GPUMaxConcurr. But, the HMR/M3R/Spark schedulers typically schedule several map tasks on each machine in the cluster. To ensure that all the tasks on a machine do not attempt to "acquire" the GPU and schedule tasks on it, we use per-machine "atomic counters" to co-ordinate between the tasks. We implement these atomic counters using ETCD [17] and ETCD's compare and swap operation. In our implementation of map, we implement a logical switch, which first checks whether the GPU is available (i.e., the atomic counter for that machine is less than GPUMaxConcurr), and then decides whether to continue executing on the CPU or to offload computation. We thus implement cooperative GPU load-balancing between tasks in all the three frameworks. This is a simple and effective solution that avoids changes to both the API and the schedulers in the frameworks.

D. Avoiding Warp Divergence

NVIDIA GPUs have a number of multiprocessors, each of which executes in parallel with the others. A Kepler multiprocessor, for example, has 12 groups of 16 stream processors. We will use the more common term, core, to refer to a stream processor. Each core can execute a sequential thread, but the cores execute in SIMT (Single Instruction, Multiple Thread) fashion; all cores in the same group execute the same instruction at the same time, much like classical SIMD processors. Code is thus actually executed in groups of threads, referred to as a warp. If there is branch divergence among the threads in a warp, i.e., if two threads in a warp execute a conditional (if..then) and follow different execution paths, then the performance of the entire warp suffers. Consequently, we either ensure the absence of conditionals when parallelizing map and reduce tasks, or choose not to parallelize the task if conditionals cannot be avoided.

Grouping of threads into warps is not only relevant to computation, but also to global memory accesses. The device coalesces global memory loads and stores issued by threads of a warp into as few transactions as possible to minimize DRAM bandwidth. Thus, when moving data into GPU memory, we reorder the data so that two parallel map or reduce threads access consecutive locations in GPU global memory.

E. Utilizing GPU Memory Effectively

GPUs have their own memory hierarchy. Each thread in a warp has its own local memory, mainly registers managed by the processing core. There is also a small software-managed data cache attached to each multiprocessor (GPU L1 cache), shared among the cores (called "scratchpad" memory or shared memory). This is a low-latency, high-bandwidth (typically over 1TB/s), indexable memory which runs close to register speeds; but it is much smaller compared to global GPU memory. Global GPU memory is off-multiprocessor memory, which can be accessed by all threads executing on the GPU, across all multi-processors. Global GPU memory also has high bandwidth, typically over 170 GB/s, but accessing it is typically slower than shared memory. Consequently, GPUs also have an L2 cache to cache contents between global memory and the processing cores; all accesses to global GPU memory go through L2 cache.

While implementing the GPU kernel, we manually store elements that are used frequently on the GPU L1 cache. This is key for iterative machine learning algorithms. We examine each loop in the iterative algorithm, and if the loop involves a computation with respect to a fixed set of data points, we store them in L1 cache. For example, in the case of K-Means clustering, every map task computes Euclidean distance to a fixed set of centroids per iteration. Consequently, at the start of each iteration, we store the current set of centroids in L1 cache.

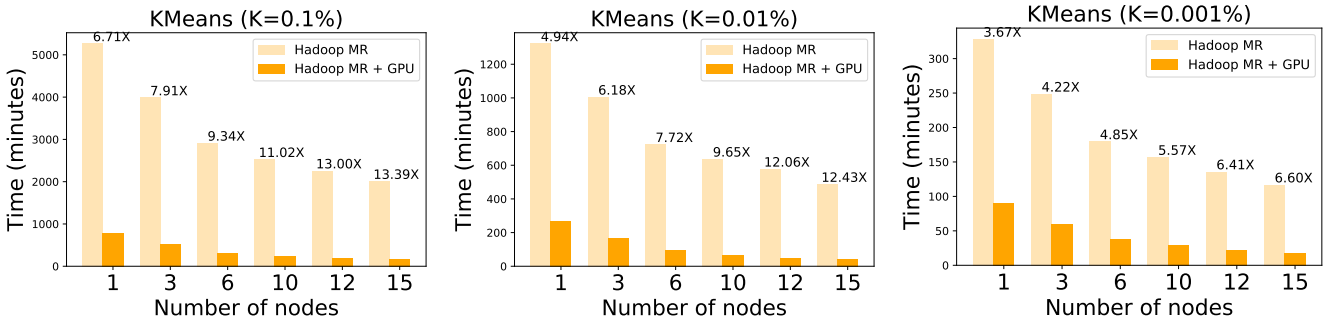
V. Evaluation

We first discuss our experimental setup and evaluation methodology, and then discuss our evaluation results for accelerating Hadoop Map-Reduce, Main-Memory Map-Reduce, and Spark.

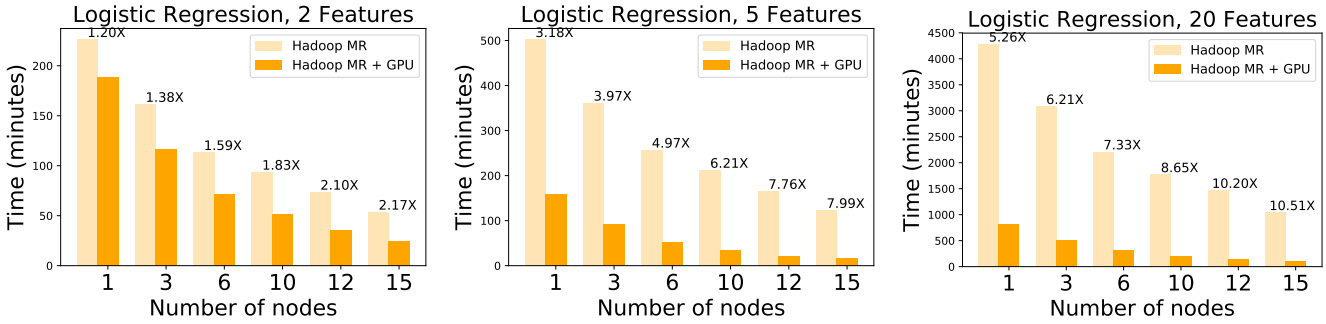
A. Experimental Setup and Methodology

All experiments, unless otherwise mentioned, are conducted on the Amazon EC2 IaaS cloud. We use EC2 G2 instances (g2.2xlarge) [4], each with one NVIDIA GRID K520 GPU with 1536 CUDA cores at a clock frequency of 800 MHz and 4GB of global GPU memory. Each instance also has 8 vCPUs, 15GB of RAM and a 60GB SSD drive.

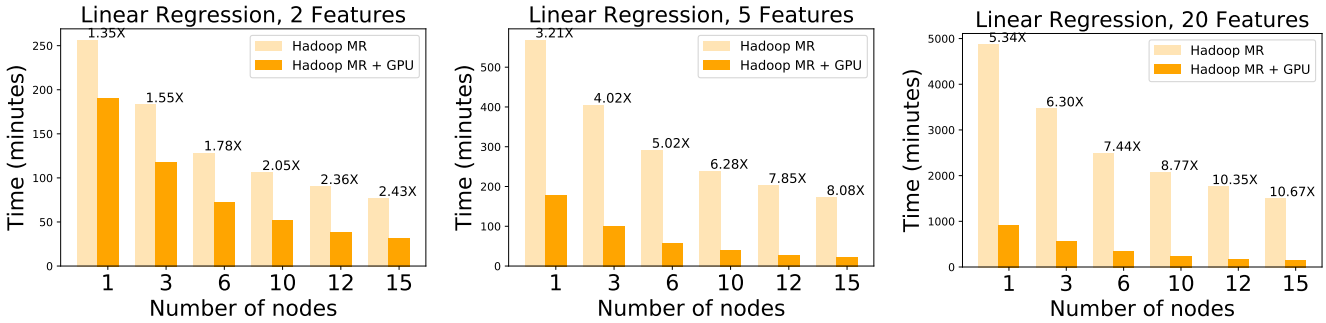
For each application, we vary the number of EC2 instances from 1 to 15, and consequently the number of vCPUs and GPUs. We use the open-source data generator included with Apache Spark [18] to generate a 100GB workload for the applications with different characteristics. We believe that using a data generator as opposed to using a single dataset enables us to vary the parameters of the workload and correlate speedups to workload characteristics. This data generator has been used extensively in industry for benchmarking Spark and various implementations of Hadoop Map-Reduce. For K-Means, we generate a single 100GB dataset where points have 20 dimensions (features), and measure the impact of using GPUs by varying the number of clusters, K. The three values of K we use are 0.001%, 0.01% and 0.1% of the number of items in the dataset. In the case of multiple linear regression, we vary the number of dimensions (features) of the data – 2, 5 and 20 dimensions. We do the same for multinomial logistic regression. For principal component analysis, we also use a 100GB dataset, and determine the top 10, 100 and 1000 principal components. We note that the parameters we



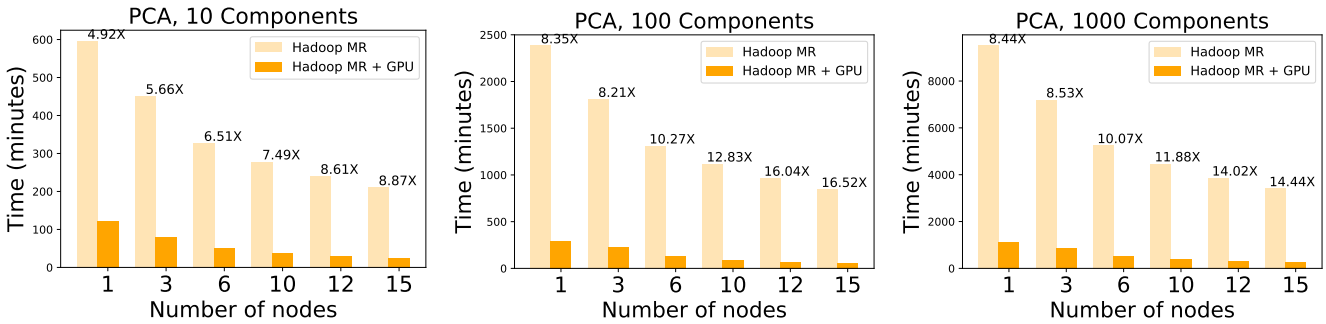
(a) Accelerating K-Means Clustering in Map-Reduce using GPUs



(b) Accelerating Multinomial Logistic Regression in Map-Reduce using GPUs

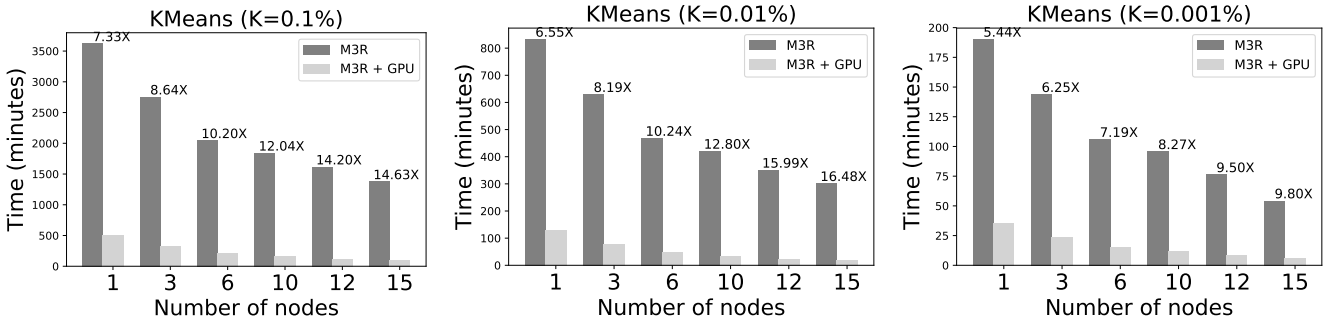


(c) Accelerating Multiple Linear Regression in Map-Reduce using GPUs

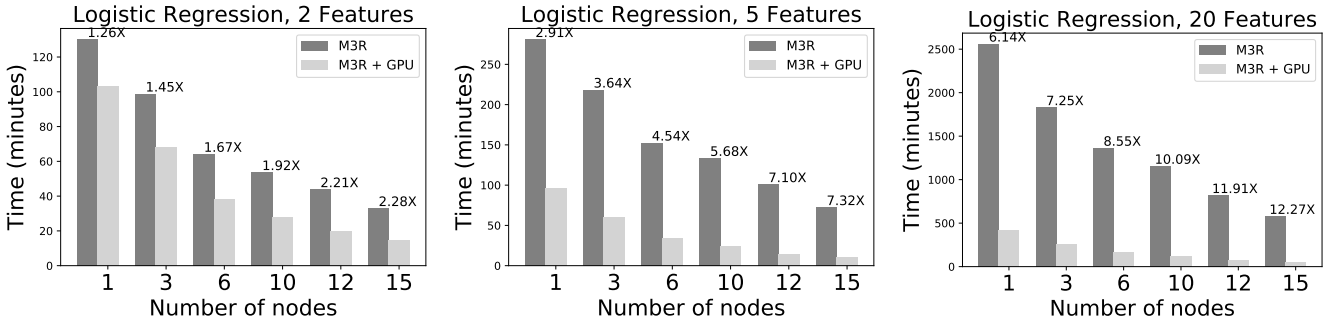


(d) Accelerating PCA through SVD in Map-Reduce using GPUs

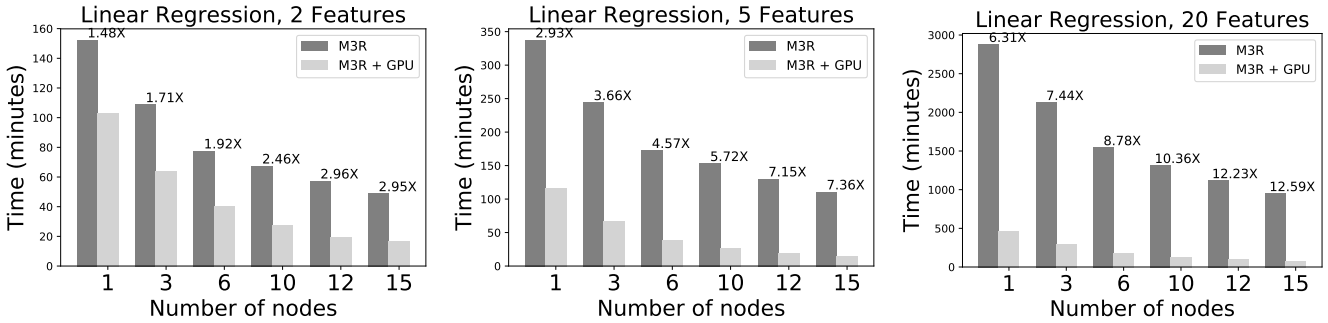
Fig. 1: Accelerating Map Reduce with GPUs



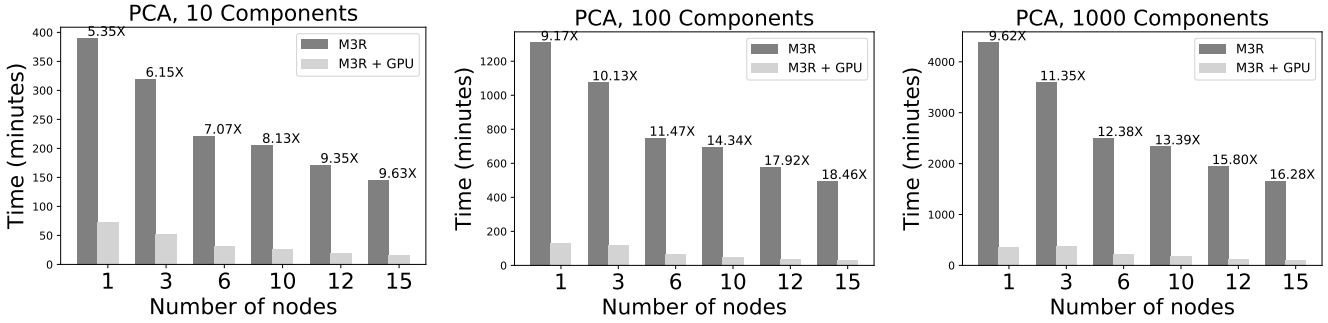
(a) Accelerating K-Means Clustering in Main Memory Map Reduce (M3R) using GPUs



(b) Accelerating Multinomial Logistic Regression in Main Memory Map Reduce (M3R) using GPUs

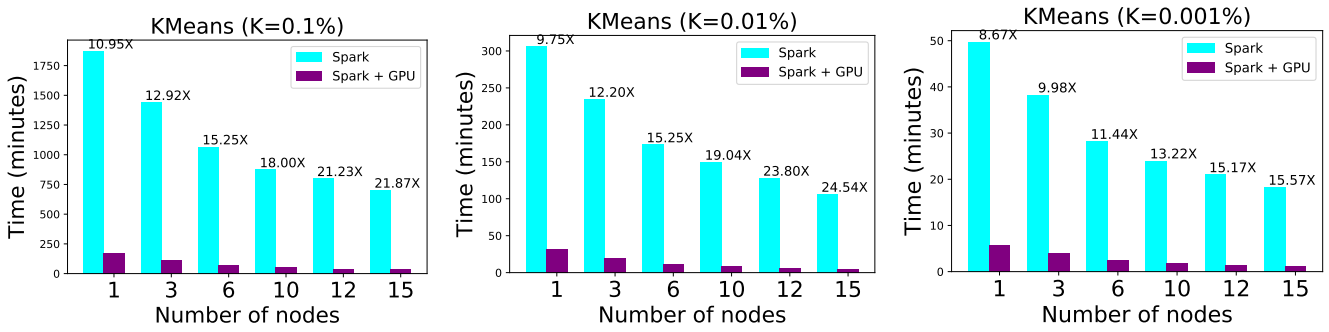


(c) Accelerating Multiple Linear Regression in Main Memory Map Reduce (M3R) using GPUs

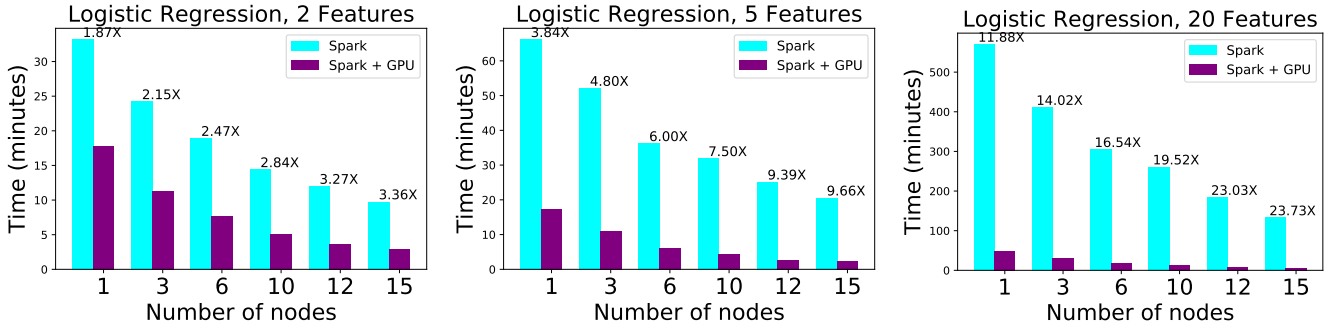


(d) Accelerating PCA through SVD in Main Memory Map Reduce (M3R) using GPUs

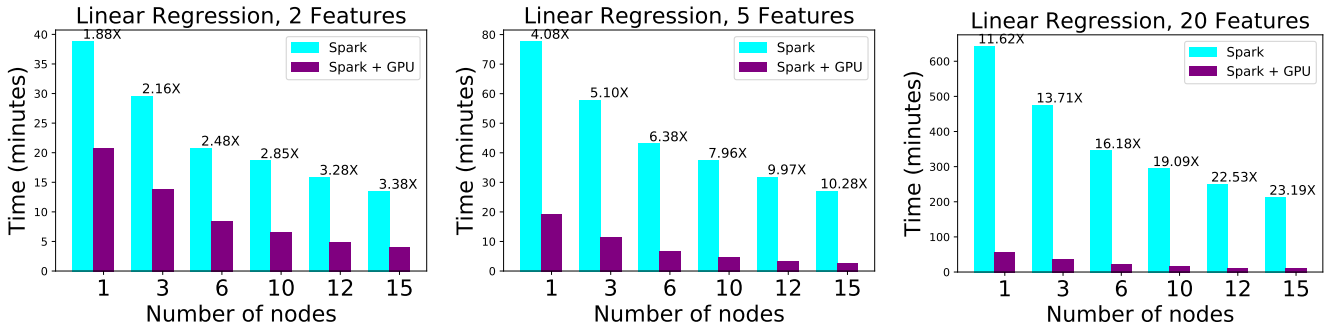
Fig. 2: Accelerating Main Memory Map Reduce (M3R) with GPUs



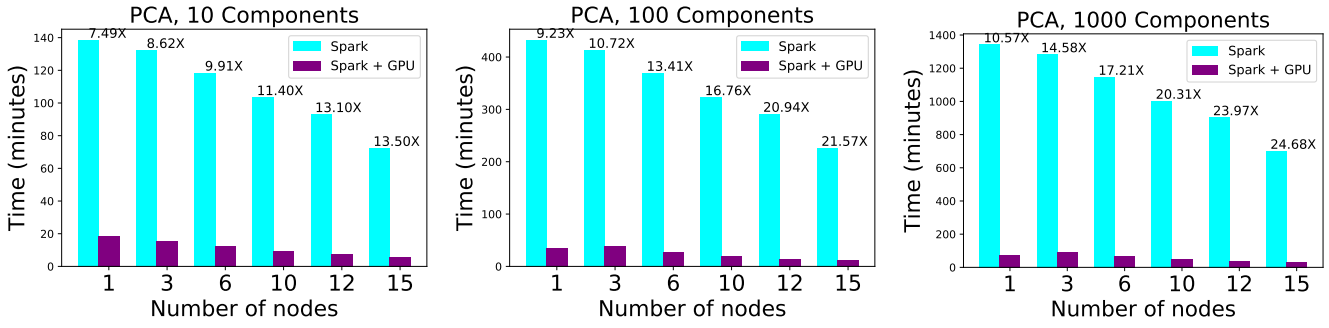
(a) Accelerating K-Means Clustering in Spark using GPUs



(b) Accelerating Multinomial Logistic Regression in Spark using GPUs



(c) Accelerating Multiple Linear Regression in Spark using GPUs



(d) Accelerating PCA through SVD in Spark using GPUs

Fig. 3: Accelerating Spark with GPUs

have chosen to vary for each application in our experiments are typical of real-world scenarios. For K-Means clustering, the maximum number of centroids used in our experiments corresponds to 0.1% of the number of data items. Similarly, many real-world datasets have tens of features.

The results of our experiments are illustrated in Figures 1–3. The reported values are averages based on 10 runs of each experiment. Each row of graphs corresponds to one application, and shows the application execution time, with and without GPUs, as a function of the size of the analytics cluster (in terms of the number of EC2 instances) using HMR, M3R, and Spark frameworks. We also report the *speedup* in execution time afforded by GPU acceleration in each case (denoted by the numbers on top of the bars). The different graphs in each row correspond to the different parameter settings we experiment with for each application.

B. Hadoop Map-Reduce (HMR)

The first set of bars (diagonal lines hatch) in Figures 1a–1d shows our results for execution time under HMR with and without GPU acceleration for all four applications. We observe the following trends:

- The performance benefits of using GPUs are clear. Except in a few cases (Logistic and Linear Regression with two features), the speedup of using GPUs is at least 2×. Even in the case of two features, where the amount of computation is less than other scenarios, the speedup is at least 30% (and often higher).
- For each application, an increase in the amount of computation increases the speedup as a result of using GPUs. Note the increase in y-axis range in each row of figures as we increase the parameter (from left to right). In the case of K-Means, increasing the number of centroids increases computation. That is, since every data point's Euclidean distance has to be computed against every centroid to find the nearest centroid, more centroids implies more computation. Consequently, the speedups are greatest when K is 0.1% of the data items in the dataset. GPUs are most effective when the proportion of computation with respect to I/O in an application is the greatest; and since the computation of Euclidean distance for a point is independent of the other points, the scenario with the most centroids results in the greatest speedup, irrespective of the size of the cluster. Similarly, for logistic regression, an increase in the number of dimensions (features) significantly increases computation. Consequently the speedups are greatest for 20 features. We see a similar trend in the case of Linear Regression and PCA through SVD.
- As expected, the speedup increases for all four applications when the number of available GPUs increases. This is because we parallelize all possible map, combine and reduce tasks, and increasing the number of GPUs decreases the execution time of the application. But, we observe that the "magnitude of increase" in speedup decreases as the cluster size increases. This is because, when the size of the dataset remains constant at 100GB, beyond a certain point, the proportion of time spent in communication, disk I/O and shuffling data increases when compared to compute.
- The application with the maximum amount of computation also has the greatest speedup. GPUs are particularly effective for matrix computations [12], [20]. Consequently, PCA (through SVD) has the most speedup among all the ML applications.

C. Main-Memory Map-Reduce (M3R)

The second set of bars (circles hatch) in Figures 2a–2d shows our results when accelerating M3R with GPUs for all four applications. We observe the following trends:

- Except in cases where compute is relatively small (Logistic and Linear Regression with two features), the speedup is significant, at least 3×, and often greater than 5×. For the case of Logistic and Linear Regression with two features, the speedup (26% – 92%), while modest, may still be beneficial cost-wise.
- In terms of absolute execution times, we find that M3R is faster than HMR, even when only CPUs are used. This is because M3R does not store intermediate results on disk.
- Since the proportion of disk I/O is reduced significantly in M3R when compared to HMR, the speedups when accelerating M3R with GPUs are generally higher than the corresponding scenarios in HMR. We observe that the use of GPUs speeds up K-Means by up to 16.5×, Logistic Regression by up to 12.3×, Linear Regression by up to 12.6× and PCA by up to 17.5×.
- Similar to HMR, an increase in the amount of computation in the workload correspondingly increases the speedup of M3R. Consequently, we observe the greatest speedup for K-Means when K=0.1%, with 20 features in logistic and linear regression, and with 1000 components for PCA.
- Although M3R eliminates a number of overheads from HMR, any map-reduce implementation involves data shuffling and network communication. When the cluster size is increased, there comes a point when the time spent on computation is smaller in proportion to communication cost. As a result, we observe from Figures 2a–2d that the speedup of using GPUs decreases and almost flatlines between 12 and 15 EC2 instances.

D. Apache Spark

The third set of bars (squares hatch) in Figures 3a–3d shows our results for accelerating Apache Spark with GPUs for all four applications. We observe the following trends:

- The trends for Apache Spark are similar to that of M3R and HMR, but the magnitude of execution times is much smaller. In general, a Spark application takes much less time than either HMR or M3R to execute, because Spark, by design, eliminates several unnecessary overheads of HMR while retaining a fault tolerance model similar to HMR.
- The speedups are significantly higher – up to 24.5× for K-Means, up to ∼24× for Logistic and Linear Regression, and 24.7× for PCA. In fact, the minimum speedup is 87% even if only logistic and linear regression with two features are

considered. This is because the I/O and shuffle overheads of Spark are much lower than that of M3R and HMR, and thus do not hinder the speedup potential of GPUs. In absolute terms, Spark with GPUs is able to crunch all the datasets in less than one hour in all cases; in many cases, in less than 30 minutes.

- The correlation between increased computation and increased speedup remains. The highest speedup for K-Means occurs in the case of K being 0.1%, logistic and linear regression with 20 features, and PCA with 1000 components.
- Despite the shuffle and I/O overheads being the lowest of all three frameworks, with a constant dataset size, their proportion relative to compute increases with an increase in cluster size. Therefore, speedups also flatline between 12 and 15 EC2 instances.

VI. Related Work

When compared to related work, this is the first paper to accelerate stock implementations of Hadoop Map-Reduce, M3R and Apache Spark on a multi-node cluster combining CPUs and GPUs, without modifying the interfaces, implementations and fault-tolerance properties of these frameworks. This is also the first paper to demonstrate that significant speedups are possible by using simple GPU acceleration strategies, without the need for complicated scheduling that only works for certain classes of workloads [15].

Mars [12] was the first project to accelerate map-reduce using GPUs. It reimplements the map-reduce programming model in C++, without fault tolerance and only works for a single physical machine. Also, Mars does not use coupled CPU-GPU execution, instead opting to execute programs on GPU only. Furthermore, the speedups reported in Mars [12] do not account for the time taken to load data into/out GPU memory; and Mars only works when the entire data set fits in GPU memory. This severely limits its applicability. MapCG [20] is a MapReduce framework to provide source code level portability between CPU and GPU. Programmers only need to write one version of code that can be compiled and executed on either CPUs or GPUs efficiently without modification. Similar to Mars, MapCG is also not distributed and is only applicable to single-machine workloads. The Phoenix [29] and its successor Phoenix++ [32] platforms from Stanford are also non-distributed, but target the implementation of map-reduce for multi-core CPUs and shared-memory multiprocessors (SMPs and ccNUMAs). Both are implemented in C++, and follow a design vastly different from that of HMR, because they target different hardware platforms. They also do not handle fault tolerance. CellMR [28] is an efficient and scalable implementation of the MapReduce framework for asymmetric Cell-based clusters. The novelty of CellMR lies in its adoption of a streaming approach to supporting MapReduce, and its adaptive resource scheduling schemes: Instead of allocating workloads to the components once, CellMR slices the input into small work units and streams them to the asymmetric nodes for efficient processing. Moreover, CellMR removes I/O bottlenecks by design, using a number of techniques, such as double-buffering and asynchronous I/O, to maximize cluster performance. The Cell processor, however, is not a GPU and has a different design from conventional GPUs – Cell is is a microprocessor intended as a hybrid of conventional desktop processors (such as the Athlon 64, and Core 2 families) and more specialized high-performance processors, such as the NVIDIA and ATI graphics-processors. [15] is the first work to consider coupled CPU-GPU execution for map-reduce and implements sophisticated scheduling, load balancing and pipelining schemes. However, it is neither distributed nor fault-tolerance. Importantly, *none of the systems mentioned above has been evaluated on more than 1GB of data.*

Dandelion [30] is a recent system that provides a unified programming model for heterogeneous systems that span diverse execution contexts including CPUs, GPUs, FPGAs, and the cloud. It adopts the .NET LINQ approach, integrating data-parallel operators into general purpose programming languages such as C# and F#. It therefore provides an expressive data model and native language integration for user-defined functions, enabling programmers to write applications using standard high-level languages and development tools. Dandelion, as mentioned, targets a different programming model than the three frameworks in this paper. Our eventual goal is to develop such an open-source infrastructure for Hadoop Map-Reduce and Spark using IBM's upcoming Lime compiler [11]. HeteroSpark [24] is a recent framework for accelerating Spark applications by providing Java RMI interfaces for integration with GPUs. Its overheads are much higher than our approach due to the use of RMI for each GPU offloading task, and also because it does not make any attempt to avoid non-coalesced memory accesses or use GPU memory bandwidth and hierarchy efficiently.

TensorFlow [1] is an open-source software library for dataflow programming across a range of tasks – it helps in improving the performance of numerical computation and neural networks and generating data flow as graphs – consisting of nodes denoting operations and edges denoting data arrays. Apache Spark/Hadoop Map Reduce are generic data processing frameworks, whereas TensorFlow is used for custom deep learning and neural network design. TensorFlow requires the user to redefine the data flow graph when the number of nodes changes; Spark and HMR, on the other hand, automatically adapt the execution to the number of nodes.

VII. Conclusion

In this paper, we examine the extent to which iterative ML algorithms implemented in Hadoop Map-Reduce, Spark and IBM Main Memory Map-Reduce can be accelerated with GPUs. We demonstrate that employing simple GPU programming practices, like using GPU memory effectively (both with respect to the memory hierarchy and by avoiding non-coalesced memory accesses) and using atomic counters to coordinate access to the GPU, can provide substantial performance speedups. We also correlate performance improvements with parameters of the iterative ML applications that increase the amount of computation, and with the size of the cluster.

We observe that maximum GPU acceleration occurs in a framework like Apache Spark with minimal disk I/O and network communication.

Notably, we have been able to demonstrate said performance gains using "basic" (and relatively inexpensive) GPUs. We expect, and have observed, performance to further increase on more expensive GPUs, such as the K40, K80, P100 and V100.

REFERENCES

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] Alex Chen, Justin Basilico, and Xavier Amatriain. *Distributed Neural Networks with GPUs in the AWS Cloud*, 2014. http://techblog.netflix.com/2014/02/distributed-neural-networks-with-gpus.html.

[3] Amazon Web Services. *Elastic Map Reduce*, 2018. https://aws.amazon.com/emr/.

[4] Amazon Web Services Inc. *EC2 Instance Types : GPU G2*, 2015. https://aws.amazon.com/ec2/instance-types/.

[5] AMPLab UC Berkeley. *MLbase: Distributed Machine Learning Made Easy* , 2018. http://mlbase.org/.

[6] Apache Hadoop. *Map Reduce*, 2018. https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[7] Apache Hadoop. *Map-Reduce Tutorial*, 2018. https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[8] Apache Hadoop Project. *YARN : Yet Another Resource Negotiator*, 2018. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[9] Apache Spark. *The MLLib Scalable Machine Learning Library* , 2018. https://spark.apache.org/mllib/.

[10] David Arthur and Sergei Vassilvitskii. K-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[11] Auerbach, Joshua and Bacon, David F. and Cheng, Perry and Rabbah, Rodric. Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. In *OOPSLA '10*, 2010.

[12] B. He and W. Fang and Q. Luo, Qiong N.K. Govindaraju and T. Wang. Mars: A MapReduce Framework on Graphics Processors. In *PACT '08*, 2008.

[13] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[14] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[15] Linchuan Chen, Xin Huo, and Gagan Agrawal. Accelerating mapreduce on a coupled cpu-gpu architecture. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 25:1–25:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[16] Contributors to Apache Mahout. *Apache Mahout* , 2018. https://mahout.apache.org/.

[17] CoreOS. *The ETCD Key Value Store*, 2018. https://coreos.com/etcd/.

[18] Databricks Inc. *Apache Spark*, 2018. https://spark.apache.org/.

[19] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[20] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. Mapcg: Writing parallel program portable between cpu and gpu. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 217–226, New York, NY, USA, 2010. ACM.

[21] JCUDA.org . *Java Bindings for CUDA*, 2018. https://github.com/jcuda/jcuda-main.

[22] Joseph Bradley, Tim Hunter and Yandong Mao . *GPU Acceleration in Databricks* , 2016. https://databricks.com/blog/2016/10/27/gpu-acceleration-in-databricks.html.

[23] Kubernetes Community. *Production Grade Container Orchestration*, 2018. https://kubernetes.io/.

[24] Peilong Li, Yan Luo, Ning Zhang, and Yu Cao. Heterospark: A heterogeneous cpu/gpu spark platform for machine learning algorithms. In *2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 347–348, Aug 2015.

[25] Mesosphere Inc. *Mesos: A Distributed Systems Kernel*, 2018. http://mesos.apache.org/.

[26] NVidia Inc. *GPU Machine Learning Applications*, 2015. http://www.nvidia.com/object/machine-learning.html (Note: Includes links to several research papers and technical blogs summarizing interesting results).

[27] NVIDIA Inc. *The CUDA Parallel Computing Platform*, 2018. https://developer.nvidia.com/cuda-zone.

[28] M. M. Rafique, B. Rose, A. R. Butt, and D. S. Nikolopoulos. Cellmr: A framework for supporting mapreduce on asymmetric cell-based clusters. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, May 2009.

[29] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[30] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 49–68, New York, NY, USA, 2013. ACM.

[31] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. M3r: Increased performance for in-memory hadoop jobs. *Proc. VLDB Endow.*, 5(12):1736–1747, August 2012.

[32] Justin Talbot, Richard M. Yoo, and Christos Kozyrakis. Phoenix++: Modular mapreduce for shared-memory systems. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 9–16, New York, NY, USA, 2011. ACM.

[33] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.