MapRDD: Finer Grained Resilient Distributed Dataset for Machine Learning

Zhenyu Li
Zhenyu.Li@warwick.ac.uk
Department of Computer Science
University of Warwick
Coventry, UK

Stephen Jarvis
S.A.Jarvis@warwick.ac.uk
Department of Computer Science
University of Warwick
Coventry, UK

ABSTRACT

The Resilient Distributed Dataset (RDD) is the core memory abstraction behind the popular data-analytic framework Apache Spark. We present an extension to the Resilient Distributed Dataset for map transformations, that we call MapRDD, which takes advantage of the underlying relations between records in the parent and child datasets, in order to achieve random-access of individual records in a partition. The design is complemented by a new MemoryStore, which manages data sampling and data transfers asynchronously. We use the ImageNet dataset to demonstrate that: (I) The initial data loading phase is redundant and can be completely avoided; (II) Sampling on the CPU can be entirely overlapped with training on the GPU to achieve near full occupancy; (III) CPU processing cycles and memory usage can be reduced by more than 90%, allowing other applications to be run simultaneously; (IV) Constant training step time can be achieved, regardless of the size of the partition, for up to 1.3 million records in our experiments. We expect to obtain the same improvements in other RDD transformations via further research on finer-grained implicit & explicit dataset relations.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**; **Distributed architectures**; • **Computing methodologies → Machine learning**;

KEYWORDS

Apache Spark; Resilient Distributed Dataset; Random Sampling; Heterogeneous Architecture; Graphical Processing Units

ACM Reference Format:
Zhenyu Li and Stephen Jarvis. 2018. MapRDD: Finer Grained Resilient Distributed Dataset for Machine Learning. In *BeyondMR'18: Algorithms and Systems for MapReduce and Beyond , June 15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3206333.3206335

BeyondMR'18, June 15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-5703-6/18/06…$15.00
https://doi.org/10.1145/3206333.3206335

1 INTRODUCTION

In the age of big data, the rate of growth of collected data is outpacing the growth of computer memory and bandwidth, as well as the processing ability needed to digest this data. At the same time, we are entering the era of exascale computing, where unprecedented amounts of data will be matched by unparalleled computing power: In this context new data analytic frameworks will be fashioned, supporting new scientific discovery and societal change.

Real-life data takes various forms, in plain text, graphs, images and videos, etc. They can be statically stored or streamed dynamically, and processed by batch and stream processing frameworks respectively. For sparse graphs, such as those seen in the analysis of social networks, graph-parallel frameworks have been developed to take advantage of extra levels of parallelism by exploiting vertices and edges in a graph.

Due to the expense of moving large amounts of data across a network, the concept of 'moving compute to data' was proposed, and realized most notably in the map-reduce paradigm. Most data analytic frameworks follow the same philosophy that executes 'closures' - enclosed functions that produce no side-effects, on stateless workers.

Memory abstraction is a key element in a data-analytic framework. The *Resilient-Distribued Dataset (RDD)* [22] is the main concept behind the Spark framework; it is a fault-tolerant memory abstraction that allows programmers to manage data across the cluster. There are similar concepts of distributed dataset and datastream abstractions in other data-analytic frameworks and it is this that makes data-analytic frameworks distinctive from traditional *Message-Passing Interface (MPI)* models for distributed programming.

Early development of data-analytic frameworks focused on dealing with each element 'once' or 'at-least-once' in batch and stream processing. Recent applications in machine learning, however, are iterative and stochastic. As a result, mainstream data-analytic frameworks can no longer fulfill the needs of machine learning algorithms, and the compute capabilities of the hardware are consistently underutilized.

Because of this, machine learning libraries seek to implement their own distributed versions. However, this has caused confusion over what the state-of-the-art implementation is, and has resulted in libraries operating in isolation, divergence in the ecosystems, and non-translatable code between libraries, which will introduce considerable costs in future code revisions. It is therefore important to have an efficient infrastructural framework for distributed memory abstractions and task execution.

In distributed machine learning, there are two categories of overhead that impede overall performance: inter-node and intra-node communications. The former consists mainly of training model synchronizations, which averages the model weights across the cluster, and are implemented by an 'all-reduce' operation in synchronous trainings. We have explored more efficient all-reduce algorithms for data-analytic frameworks in previous work [12]. It is the data transfer between the hard-disk, main memory and device memory, which is the focus of this paper.

This research encompasses the memory abstraction (RDD) and memory management system of the Spark framework. The primary deficiency of the RDD is the synchronous sequential-access. Synchronous access patterns require the parent dataset to be fully loaded or partially loaded up to maximum memory capacity before computation; and sequential-access only permits record iteration one-by-one. It is apparent that such a design does not utilize more advanced asynchronous data transfers in modern computer architectures. Moreover, machine learning, or deep learning in particular, favours stochastic and iterative algorithms, since it is easier to get an estimate from sample data when the dataset is too large for complete analysis. Having to iterate through the dataset in order to sample a subset is a waste of processor cycles and memory usage. This is why more state-of-the-art machine learning libraries do not favour Spark as the distributed platform. A more in-depth analysis for the design of the RDD can be found in Section 3.

The root of the problem is the coarse granularity of the dataset. RDDs are split in partitions, which determine the level of parallelism and granularity of data dependencies. There are no record-wise relations between the parent and child datasets, therefore operations on individual records in the child dataset cause the entire parent dataset to be materialized.

The contributions of this paper are as follows:

- We describe the current design of the Resilient Distributed Dataset (RDD): (I) The organization of an RDD, in terms of internal structures and dependencies; (II) The computation mechanism of RDDs in in-memory mode and on-disk mode.
- We discuss the inefficiency of sampling from an RDD through iterating the entire parent dataset; we discover the source of the problem to be the coarse-granularity and sequential-access of the dataset.
- We discuss the use of RDDs on accelerators. We describe the starvation of accelerators due to the high memory pressure in a scaled-up cluster with high compute-to-memory ratio; and existing solutions do not solve the problems with exhaustive data loading and high memory pressure in sampling.
- We present the design of the new MapRDD, which exploits the record-wise relation between the parent and child datasets during *map* transformations, and permits random-access to individual records in the child dataset through computing the chain of dependent records. Random-access in the new MapRDD enables data sampling without computing the entire parent dataset.
- We discuss the mapping of dataset partitions to accelerators, and the imbalance between the partition size and CPU utilization. We propose a parallel sampling algorithm to make use of the idle processors for sampling.
- We present the implementation of a new MemoryStore for the new MapRDD, which organizes the dataset at the record level, and manages data sampling and data transfers asynchronously.
- We use the ImageNet dataset to demonstrate that the initial data loading can be eliminated by comparing the sampling performance with the original MapPartitionsRDD and the new MapRDD; the CPU processing cycles and memory usage can be reduced by more than 90%, allowing other applications to be run simultaneously.
- We also demonstrate through the ImageNet dataset that the size of a single partition in the original MapPartitionsRDD is limited to 4GB; the new MapRDD removes the constraint by managing data in records instead of blocks.
- We train AlexNet [11] with the ImageNet dataset on an NVidia Tesla K80 GPU. We demonstrate that the data sampling and data transfer can be totally overlapped with training on the GPU with the new asynchronous MemoryStore. We demonstrate a 4x speedup for up to 20% of the ImageNet dataset, in GPU training with the new MapRDD and the new MemoryStore, compared with the original MapPartitionsRDD. We also show a constant training step time with the new MapRDD, regardless of the size of the partition, for up to 1.3 million records in the ImageNet dataset.

The remainder of this paper is organized as follows: Section 2 provides an overview of distributed data-analytic frameworks and machine learning libraries; Section 3 explores the underlying structures and mechanisms for the Resilient Distributed Dataset (RDD), and how it is inefficient for machine learning and heterogeneous environments; a new design called *MapRDD* for *map* transformations and the implementation of a new asynchronous MemoryStore are inroduced in Section 4, which are evaluated against the original *MapPartitionsRDD* on the ImageNet dataset in Section 5; we conclude that the new *MapRDD* excels in minimizing unnecessary CPU processing cycles and memory usage, and also maximizes GPU utilization. Through this research, we believe that the other dataset transformations (besides *map*) can achieve the same efficiency with further research on finer-grained implicit/explicit dataset relations.

2 RELATED WORK

For data analytics, most research concentrates on the higher-level distributed frameworks and machine learning libraries, rather than the lower-level operating systems and compilers, because portability, robustness and fault-tolerance are considered more important for data processing tasks.

2.1 Data processing frameworks

Data processing frameworks can be categorized in terms of application: (I) Batch processing; (II) Stream processing; (III) Graph analytics.

Batch processing is concerned with running a series of independent tasks, and it is often employed for processing large volumes of data. MapReduce [3] is one of the early distributed batch-processing frameworks, where each element in the dataset is passed through a *Map-Reduce* pipeline. MapReduce had been superseded by more flexible data-flow models using for example, Directed-Acyclic-Graph

Listing 1: Code snippet from RDD.scala: map() and mapPartitions() Syntax

```
1  def map[U: ClassTag](f: T => U): RDD[U] = withScope {
2      val cleanF = sc.clean(f)
3      new MapPartitionsRDD[U, T](this, (context, pid, iter)
           => iter.map(cleanF))
4  }
5  def mapPartitions[U: ClassTag](
6      f: Iterator[T] => Iterator[U],
7      preservesPartitioning: Boolean = false): RDD[U] =
           withScope {
8      val cleanedF = sc.clean(f)
9      new MapPartitionsRDD(
10       this,
11       (context: TaskContext, index: Int, iter:
             Iterator[T]) => cleanedF(iter),
12       preservesPartitioning)
13 }
```

engines, see Dryad [9] and Spark [22], and more recent Stateful-Directed-Graph engines, such as that seen in Naiad [17].

Stream processing is concerned with handling continuously generated data in real-time. Modern stream processing frameworks, such as Apache Storm [20] and Apache Flink [2], share the same data-flow execution models seen in batch processing, with the addition of event-triggered mechanisms.

Graph analytic frameworks, such as Pregel [14], GraphLab [13] and PowerGraph [6], are designed specifically to deal with graph data, especially graphs with high irregularity and sparsity. They take advantage of the underlying edges and vertices in the data to achieve higher-degree of parallelism.

2.2 Machine Learning Libraries

Caffe [10], Theano [18] and Tensorflow [1] are the most popular machine learning libraries, among which Caffe and Theano are standalone implementations. Although Tensorflow was meant for distributed machine learning, [1] mostly describes how a Tensorflow application runs on a single node, and the documentation provides little explanation of how to manage distributed data efficiently. This is also due to the fact that the Tensorflow API is still under active developement.

SparkNet [16] is a toolkit that provides the missing link to supporting machine learning libraries (Caffe and Tensorflow) on Spark. But ironically, the bulk of data is pre-processed and down-sampled so that the data can fit in main memory in order to run on Spark, which defies the sole purpose of the Spark framework. The data is also pre-shuffled structurally rather than sampled from a probability distribution. In summary, SparkNet has significant limitations in its current form.

3 BACKGROUND

The *Resilient-Distributed-Dataset (RDD)* memory abstraction is the key concept underpinning the Spark framework. In this section, we explore in detail the underlying structures and working mechanisms of the RDD, and its application in machine learning and heterogeneous environments.

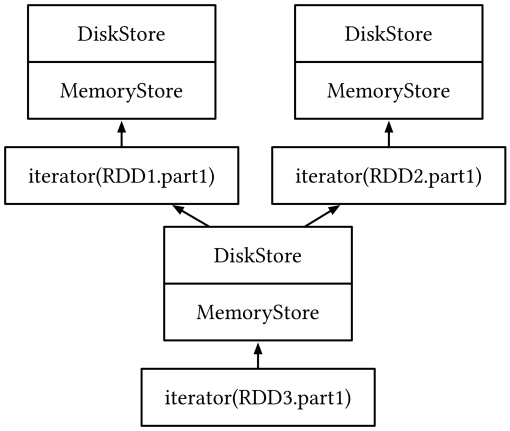

Figure 1: Computation of partition 1 in RDD3 that depends on partition 1 in RDD1 & RDD2

3.1 Parallelism

The fundamental unit of an RDD is a partition that describes a subset of the dataset, rather than the elements in the partition. When a *map* function is applied to the dataset, tasks are created for each partition. Therefore, the number of tasks is the number of partitions, so is the level of parallelism. Memory management is also organized in terms of partitions, as such, a data-block unit belongs to a single partition.

This detail turns out to be crucial to understanding the performance difference between *map()* and *mapPartitions()* transformations, as it had been recognized that there are discrepancies between the two [19] [15]. As shown in lines 3 & 11 of Listing 1, the user-function is applied to the entire partition as a single task, where *iter* is an iterator for the elements in the partition, and the difference is whether the user-function takes an element or an iterator as input, but they create the same number of tasks/threads. As demonstrated by Lester Martin [15], *map()* transformations can lead to slower performance than *mapPartitions()* transformations, if some helper objects are created for every element, but parallelism does not contribute to the performance difference.

3.2 Dependencies & Computations

A partition is the basic unit of an RDD, and dependencies describe the relationships between partitions of the parent and the child RDD. There are two types of dependencies: Narrow-Dependency and Shuffle-Dependency. For narrow dependencies, a child partition depends on a small number of partitions from the parent RDD. For shuffle dependencies, on the other hand, a child partition depends on a large number of partitions in the parent RDD.

The computation of an RDD is delegated to the MemoryStore or the DiskStore through the process of unrolling, in which the MemoryStore or the DiskStore iterates through the elements in a given partition, which is a chained-action that causes all the

Algorithm 1: Simplified illustration for unrolling an RDD partition

```
Data: source iterator of partition p
Result: output iterator of partition p
if Use Memory then
    while source.hasNext & Enough Memory do
        output.add(source.next);
        Reserve memory if needed;
    end
    if !source.hasNext then
        Return completely unrolled output iterator
    else
        Return partially unrolled output iterator
    end
else
    Unroll source iterator to file;
    Return file stream of the memory-mapped file;
end
```

dependent partitions to be computed if not already. As illustrated in Figure 1, the computation of partition 1 in RDD3 causes the materialization of partitions in the parent RDDs 1 & 2. In a sense, the Spark framework is essentially a distributed memory system.

The mechanism of the MemoryStore or DiskStore during computations is shown in Algorithm 1. Depending on whether the memory or the disk is used, the partition is either unrolled until the maximum memory is reached or written directly to disk, and this process is synchronous. What is interesting is how it handles the data that exceeds the memory limit. If disk is used, it must first write the entire content to the disk, then returns a memory-mapped image of the file. Else, the memory store must release the references to the previously unrolled elements. If the user still keeps a reference to the data (i.e., memory cannot be reclaimed by the garbage collector), an out-of-memory error is raised.

3.3 Sampling

Having shown how the MemoryStore and DiskStore handle data, we can now understand why sampling data from an RDD is inefficient. By invoking RDD.sample(), a new RDD is created by iterating through the entire parent RDD. Although drawing a sequence from a probability distribution is expensive, and efforts had been made to minimize it by using a method called Gap-Sampling [5], a much greater cost comes from the materialization of the entire parent RDD and the memory pressure when there is not sufficient physical memory to hold the data as discussed above.

The root of the problem is the granularity of the RDD (i.e., in partitions instead of records) and the sequential-access (i.e., as opposed to random-access). Sampling only requires a subset of the dependent partition, therefore it is not efficient to compute the entire partition.

There is no easy solution to the problem, because there exists no explicit relation between the records in the parent and the child dataset, nor even between the parent-child partitions in the case of a *Shuffle-Dependency*. The state of the child dataset is entirely undetermined.

3.4 RDDs on Accelerators

Machine learning using accelerators, such as Graphical Processors (GPUs), has become the main trend in recent years. Accelerated clusters have scaled-up and concentrated processing power, as opposed to a scaled-out cluster with less computationally intensive nodes. This has several significant implications on the practicalities of RDDs. As the compute-to-memory ratio is higher, applications run on fewer nodes with less main memory. With less main memory comes higher memory pressure, and data is more likely spilled to disk storage. Since the total device memory must be less than or equal to the main memory, there is even more stress on the device memory. This results in the starvation of accelerators.

There exist GPU implementations of the RDD abstraction [7] [21], which takes care of the data management between the CPU and the GPU, and the mapping of data to GPU kernels. However, it does not solve the fundamental issues concerning how the data is loaded, nor does it improve sampling efficiency. Moreover, mapping data to GPU kernels is very restrictive to programmers, since users cannot utilize the interface provided by the machine learning libraries. It is more practical to provide a handle to the GPU data, and leave the choice of programming interface and library to the programmers.

4 METHODOLOGY

In Section 3 we have explored in-depth how an RDD works, and the issues for the use of RDDs on accelerators and machine learning applications. We summarize that the RDD is inefficient for machine learning due to the coarse granularity and the synchronous sequential-access of the dataset. The objective is efficient handling of data that exceeds physical memory capacities for heterogeneous architectures, with an application for stochastic processes. We present the design of the new **MapRDD**, which exploits the implicit relations between data records in *map()* transformations; we describe the design of *MapRDD* in the remainder of this section.

4.1 New MapRDD vs. MapPartitionsRDD

As explained in Sub-section 3.3, the dataset granularity is limited by the non-explicit relation between the parent and the child datasets. There is, however, an implicit relation between the records by the *map()* transformation due to the syntax of the *map()* function $map : f(A) \rightarrow B$.

In the current implementation of Spark, both *map()* and *mapPartitions()* transformations produces a MapPartitionsRDD, in which the data granularity is kept at the partition level, such that it is consistent with other data transformations (such as *sortByKey()*, *groupByKey()*, *cogroup()*, etc.).

We introduce a new *MapRDD* that exploits the implicit relations of $map : f(A) \rightarrow B$. Figure 2 shows a layout of the architectural differences of the original RDD implementation and the new asynchronous *MapRDD* implementation. From the top down, they are: (I) User interface, an iterator that draws items from the dataset; (II) Memory abstraction that describes the dataset; (III) Memory/Disk

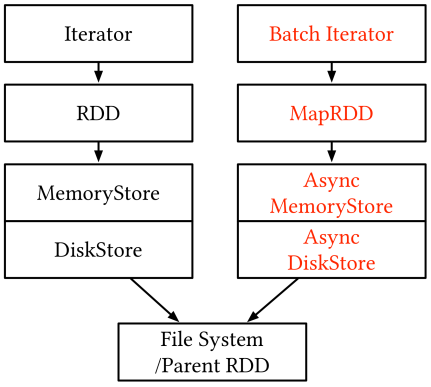

Figure 2: Overall Architecture. Left: Original RDD Implementation; Right: New Sample RDD Implementation

Store that manages data objects in memory and on disk; (IV) Parent dataset, or the underlying file system at the source level. The new *MapRDD* is an extension to the original RDD at all levels except for the file system. It preserves backward capabilities to the original RDD, as such the new RDD can be the base/parent RDD of the original RDDs.

4.2 Random-Access & Sampling

The RDD supports an interface that iterates the elements in a one-by-one manner, rather than in a randomly-accessible fashion. The primary reason for this is that the state of the dataset is undetermined. The other reason for an iterator interface is that the iterated records can be safely discarded and recycled by the garbage collector; whereas in a randomly-accessible collection (e.g., arrays), memory cannot be recycled as each of the records is referenced.

With the implicit relation of *map* transformations, the size of a child *MapRDD* is known to be the same as its parent. Therefore, random-access to individual records is possible by applying the transformation function to the chain of dependent records.

We have seen in Sub-section 3.3 how sampling is in-efficient by iterating the entire dataset. With the random-access made possible by record-wise granularity in the new *MapRDD*, it is now possible to draw sample records randomly without materializing the complete dataset.

In addition, we extend the *iterator* interface to draw batches of records. It not only permits direct sampling from the current dataset, which bypass the creation of a child dataset; but also provides opportunities for the sampling algorithm and data loading to be carried out asynchronously.

4.3 Parallel Sampling for Large Partitions

The sampling process consists of a series of independent tests from a probability distribution with known parameters. The cost of computing the probabilities is relatively expensive and therefore Spark has been seeking algorithmic accelerations. An example of this is Gap-Sampling [5], as mentioned previously.

Sequential sampling is implemented in Spark, since the number of tasks in Spark is determined by the number of partitions (as explained in Sub-section 3.1), and each task takes a single processor by default. For heterogeneous architectures, the dataset is partitioned by the number of accelerators, therefore there are far fewer but larger partitions. Sequential sampling large partitions is not efficient due to the imbalance in the number of partitions and the number of CPU cores.

Listing 2: Simplified implementation of parallel sampling

```
1 def parallelSampling(partitionSize, sampler): Array[Int]
    = {
2   (0 until partitionSize).par.map(i => {
3     if (sampler.sample()) (i, 1)
4     else (i, 0)
5   }).filter(e => e._2 > 0)
6 }
```

With the size of the child dataset known in the new *MapRDD*, a parallel sampling algorithm can be implemented as shown in Listing 2. The *parallelSampling()* function takes the partition size and the sampler as arguments, and produces a parallel collection of indices from 0 to the partition size; for each of the indices, the sampler is invoked to decide if the index should be sampled (i.e. 1 for positive, 0 for negative); a final set of sample indices is produced by filtering the sampler outputs (i.e. greater than 0).

4.4 Asynchronous MemoryStore

The MemoryStore and the DiskStore are core components of the Spark framework, where the computation and the memory management of the RDD take place; as described in Sub-section 3.2. We have also discussed how a synchronous MemoryStore can be in-efficient for modern computer architectures and for datasets that exceed physical memory capacities, especially for stochastic applications.

Algorithm 2 illustrates a simplified implementation of an asynchronous MemoryStore. The workhorses of the MemoryStore are the *ReadThread* and the *WriteThread* that run asynchronously in the background while the executor computes the user-function on the next batch. As the user-function invokes the *nextBatch()* function, it immediately returns the pre-fetched batch, and signals the *ReadThread* to prepare the next batch. The *ReadThread* first samples a list of records to be computed, and cross-references any records that may have been buffered in memory or saved to disk. It then computes the record and its dependent records from the parent dataset, and reads the saved records from disk. Lastly, the *ReadThread* signals the *WriteThread* before setting the value of the *NextBatch*. The *WriteThread* in turn checks if the buffer has exceeded its limit and writes any unsaved records to the disk.

5 EVALUATION

We evaluate the new MapRDD on the ImageNet [8] dataset with AlexNet [11] on Caffe. The dataset consists of 1.3 million resized images (256x256 pixels), which is 19GB uncompressed. The experiment setup is explained in Sub-section 5.1, and the results are evaluated in terms of overall runtime, CPU utilization and GPU

Algorithm 2: Asynchronous implementation of the Memory-Store

```
1:  AllRecords ← Collection[Id, Dependency]
2:  Saved ← Map[Id, FilePath]
3:  Buffered ← Map[Id, Record]
4:  NextBatch ← wait for ReadThread
5:  procedure ReadThread
6:      loop
7:          Wait for signal
8:          batch ← Sample(AllRecords)
9:          toRead ← batch ∩ Saved
10:         inMemory ← batch ∩ Buffered
11:         toCompute ← batch − toRead − inMemory
12:         Read(toRead)
13:         Compute(toCompute)
14:         Buffered.add(toCompute, toRead)
15:         NextBatch ← batch
16:         Upload(NextBatch)
17:         Signal WriteThread
18:     end loop
19: end procedure
20: procedure WriteThread
21:     loop
22:         Wait for signal
23:         spill ← Spill(Buffered)
24:         toWrite ← spill ∩ Buffered
25:         Write(toWrite)
26:         Saved.add(toWrite)
27:         Buffered.remove(spill)
28:     end loop
29: end procedure
30: function nextBatch
31:     ret ← NextBatch
32:     Signal ReadThread
33:     return ret
34: end function
```

utilization; these results can be found in Sub-sections 5.2, 5.3 & 5.4 respectively.

5.1 Experimental Setup

The experiments are carried out on a standalone workstation with a NVidia Tesla K80 card (only a single GPU is used), so that there is no model synchronization overhead; the specification of which is listed in Table 1.

The user code is a modified version of the SparkNet [16] toolkit. The main loop implemented with the original Spark API is shown in Listing 3, and the new implementation with the new MapRDD is shown in Listing 4. The main differences are the main loop inside the *foreachPartition* structure in line 4, and the new batch interface in line 5 of Listing 4.

We run the experiments with various data sizes and memory cache settings, which are listed in Table 2; the batch size is set to the default value (i.e. 256) in the reference Caffe training model [4]; all experiments are repeated 5 times. In the original Spark

Table 1: System Configuration

CPU	Intel(R) Xeon(R) CPU E5-2687W v3 @ 3.10GHz
Memory	DDR4, Capacity:128 GB, Speed: 2400MHz
Primary Disk	Samsung SSD 850 PRO 512GB
Secondary Disk	TOSHIBA HDWE160 (6TB, 7200RPM)
Accelerator	NVidia Tesla K80

Listing 3: Main Training Loop with current Spark API

```
1 val trainRDD = new MapPartitionsRDD()
2 for (i <- 0 until iters) {
3   val sampleRDD = trainRDD.sample()
4   sampleRDD.foreachPartition(
5     trainIt => {
6       solver.step(trainIt)
7     })
8 }
```

Listing 4: Main Training Loop with new MapRDD

```
1 val trainRDD = new MapRDD()
2 trainRDD.foreachPartition(
3     batchIt => {
4         for (i <- 0 until iters) {
5             solver.step(batchIt.nextBatch)
6         }
7     }
8 })
```

implementation, the size of a partition cannot exceed 4GB, as the indexing limit is set to the maximum value of an 32-bit integer. Since we are consolidating the data into a single partition for the GPU, we are only using 20% of the ImageNet dataset. For caching methods, the 'Memory & Disk' mode uses memory as much as possible until it spills to the disk; the 'Disk-Only' mode does not cache in memory, to simulate a short-of-memory situation; the 'Async' mode is the new mode that saves data asynchronously in the new MapRDD.

5.2 Overall

Table 3 lists the runtime results corresponding to the experimental settings in Table 2. The loading time includes all the time spent from the initialization of the application untill the first training step; the average step time is the averaged runtime for each training step. Figures 3 & 4 are direct comparisons of the loading time and step time for the synchronous method in 'Memory & Disk' mode and the asynchronous method with the new MapRDD.

For the first set of experiments (i.e., 1-4) that run in 'Memory & Disk' mode, the initialization takes significant time (i.e., more than 13 minutes for 5% of the dataset) until the training finally begins, which could have been used to train for 180-280 steps. As shown in Figures 3 & 4, both the loading time and the step time increase near

Table 2: Experimental Settings

Experiment	Data Size	Cache Method	Iterations	Batch Size
1	5%	Memory & Disk	20	256
2	10%	Memory & Disk	20	256
3	15%	Memory & Disk	20	256
4	20%	Memory & Disk	20	256
5	5%	Disk Only	20	256
6	10%	Disk Only	20	256
7	15%	Disk Only	20	256
8	20%	Disk Only	20	256
9	5%	Async	20	256
10	10%	Async	20	256
11	15%	Async	20	256
12	20%	Async	20	256
13	50%	Async	20	256
14	100%	Async	20	256

Table 3: Overall Runtime

Experiment	Loading Time (sec.)	Average Step Time (sec.)
1	823.4	4.6
2	1530	6.5
3	2309.5	8.5
4	3252.9	11.3
5-8	failed	failed
9	4.2	2.7
10	4.2	2.7
11	4.2	2.7
12	4.2	2.7
13	4.2	2.7
14	4.2	2.7

linearly as the size of the partition increases; the gradient starts to grow after 15% of the ImageNet dataset (i.e., 256k records), caused by the memory pressure; this is discussed in Sub-section 5.3.

For the third set of experiments (i.e., 9-14) in asynchronous mode, the loading time is almost negligible compared with the loading time in the 'Memory & Disk' mode; a 1.7-4.2x speedup is observed in training steps for up to the partition size limit of 4GB. Both the loading time and the step time are constant in spite of the increase in data size (see Figures 3 & 4). This demonstrates that the loading time can be totally avoided by lazy-loading of data records; the asynchronous sampling and memory transfers by the new MemoryStore (see Algorithm 2) are effective, which kept the step time constant, even for a full size dataset such as the ImageNet on a single machine.

For the experiments run in 'Disk Only' mode (i.e., 5-8), they failed with the output size exceeding the maximum value of the integer type (i.e., 2^{32}) while trying to write the partition to the disk; this is because the images are expanded 4 times in size as every pixel byte is converted to a 4 byte floating-point number, and a single file cannot exceed the limit of 4GB (by the limit of 2^{32} bytes). The memory usage is also reflected in Sub-section 5.3. This implies that

Table 4: CPU Resource Utilization

Experiment	Peak Memory (Loading) (GB)	Peak Memory (Training) (GB)	CPU (Loading) (%)	CPU (Training) (%)
1	48	58	70	6
2	48	76	70	6
3	51	88	70	6
4	50	89	70	6
5-8	failed	failed	failed	failed
9	2	2.5	6	11
10	2.5	2.5	6	11
11	2.5	2.5	6	11
12	2.5	3.5	6	11
13	2.5	3.5	6	11
14	3.5	3.5	6	11

Table 5: GPU Resource Utilization

Experiment	Average block time per step (sec.)	Average compute time per step (sec.)	GPU(%)
1	2.58	2.07	44.52%
2	4.48	2.07	31.60%
3	6.7	2.07	23.60%
4	9.6	2.07	17.74%
5-8	failed	failed	failed
9	0	2.07	100.00%
10	0	2.07	100.00%
11	0	2.07	100.00%
12	0	2.07	100.00%
13	0	2.07	100.00%
14	0	2.07	100.00%

the dataset must be split into small partitions, or the application would fail. For large data items, such as high-resolution images and videos, a partition may contain very few items limited by the size of 4GB, but in large quantity. Sampling from millions of partitions and mapping these partitions to devices is not efficient. Since the new MemoryStore (see Algorithm 2) manages data in a per-record fashion, it no longer poses a size limit on the partitions, and it is more suitable for managing large items therefore.

5.3 CPU Resource Utilization

Table 4 lists the peak memory and CPU usage during loading and training, corresponding to the experiments listed in Table 2.

In terms of CPU memory, peak memory usage is much higher for the synchronous method than the asynchronous counterpart, as expected. As shown in Figure 5, the committed memory (i.e., size of the JVM heap) during training steps increases rapidly as the partition size increases for the synchronous method (i.e., experiments 1-4), whilst the memory usage of the asynchronous method (i.e.,

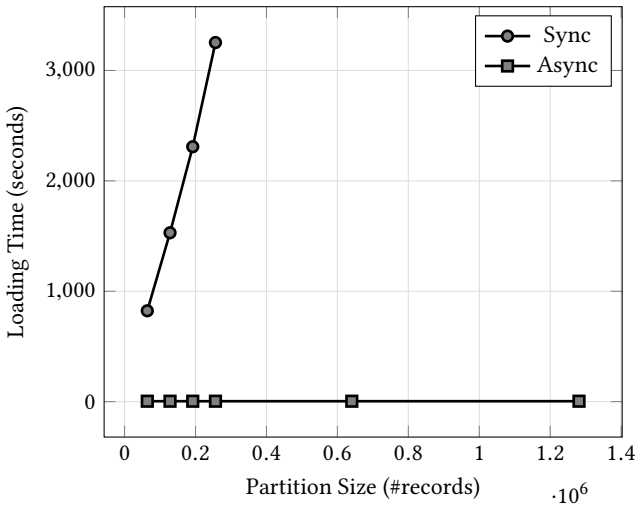

Figure 3: Average loading time across different partition sizes for synchronous (experiments 1-4) and asynchronous (experiments 9-14) methods

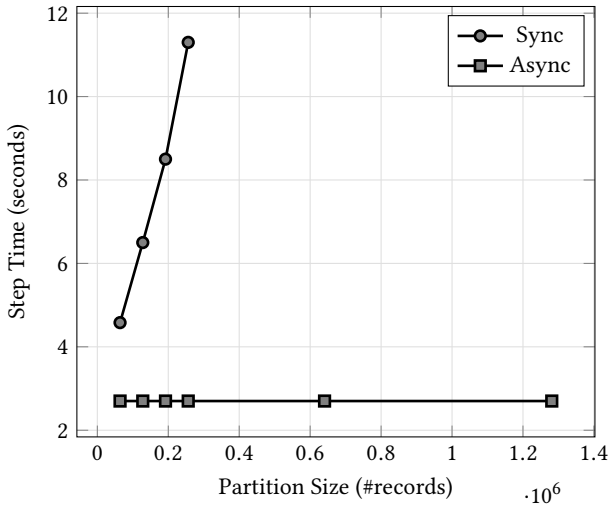

Figure 4: Average step time across different partition sizes for synchronous (experiments 1-4) and asynchronous (experiments 9-14) methods

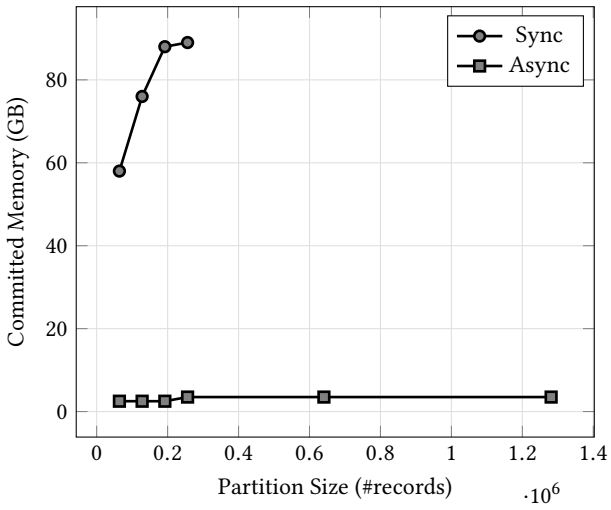

Figure 5: Average committed memory during training, across different partition sizes, for synchronous (experiments 1-4) and asynchronous (experiments 9-14) methods

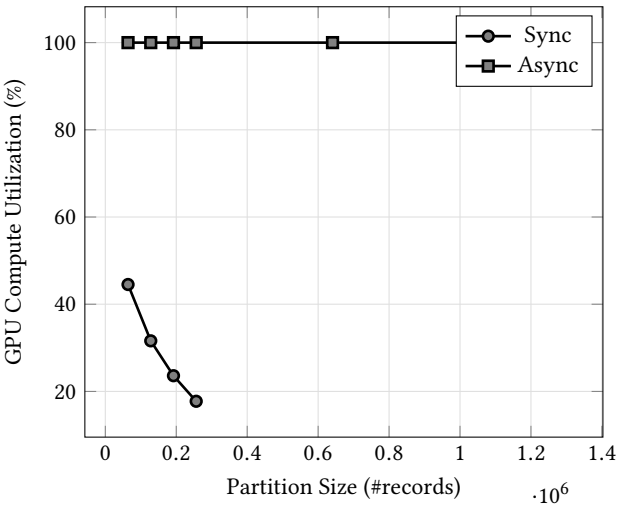

Figure 6: Average GPU compute utilization during training, across different partition sizes, for synchronous (experiments 1-4) and asynchronous (experiments 9-14) methods

experiments 9-14) is near constant. For experiment 4, the size of the heap of the Java Virtual Machine has almost reached the limit of the physical memory capacity, which cannot grow any further, therefore causing the loading and training to slow down as seen in Sub-section 5.2. In our experiments, the peak memory usage of the asynchronous method is reduced by 96% during training steps compared with the synchronous method.

In terms of CPU processing cycles, the usage is stable for both synchronous and asynchronous methods. During loading, the synchronous method takes up a significant amount of CPU cycles (as much as 70%), which is freed up by the asynchronous method (to only 6%). During training steps, the parallel sampling algorithm (see Sub-section 4.3) makes better use of the free CPU cycles (i.e., CPU utilization rises from 6% to 11%) while the majority of the computation is delegated to the GPU.

5.4 GPU Resource Utilization

Table 5 lists the average block time, the compute time, and the GPU utilization during training steps corresponding to the experiments

listed in Table 2. Figure 6 draws direct comparisons of GPU utilization during training steps for the synchronous (i.e., experiments 1-4) and asynchronous (i.e., experiments 9-14) methods.

The average compute time per training step is the same for both synchronous and asynchronous methods across different sizes of the dataset, as the batch size is constant. For the synchronous experiments, the block time (mainly consisting of data sampling and data transfer) contributes to the low GPU utilization, which drops exponentially as the partition size increases. For asynchronous experiments, the block time is negligible and the GPU functions near 100% of the time, because the data sampling and data transfer on the CPU is entirely overlapped with the compute time on the GPU.

6 CONCLUSION & FUTURE WORK

In this work, we have explored in depth how the Resilient Distributed Dataset (RDD) works and how it is largely obsolete in present day machine learning applications. We identified that the source of deficiency originates from the coarse granularity and synchronous sequential-access of the dataset.

We present the new *MapRDD*, an extension to the Resilient Distributed Dataset (RDD) for *map* dataset transformations, and the new complementary asynchronous MemoryStore. Individual records in the child *MapRDD* can be accessed randomly and lazily. The data sampling and the data transfers are managed asynchronously.

Through the experiments on the ImageNet dataset over different caching methods and data size settings, it is demonstrated that: (I) The initial data loading phase is redundant and can be completely avoided; (II) Sampling on the CPU can be entirely overlapped with the training on the GPU to achieve near full occupancy; (III) CPU cycles and memory usage can be reduced by more than 90% to allow other applications to run simultaneously; (IV) Constant training step time can be achieved, regardless of the size of the partition, for up to 1.3 million records in our experiments.

The *MapRDD* only applies to *map* transformations, but the same performance improvement is expected if the same techniques are applied to other transformations (e.g., flatMap, join, cogroup, etc.). This motivates our further research in exploring implicit and explicit descriptions of the dataset relations.

ACKNOWLEDGMENT

This research is supported by Atos IT Services UK Ltd and by the EPSRC Centre for Doctoral Training in Urban Science and Progress (grant no. EP/L016400/1).

REFERENCES

[1] Martín 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. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 265–283. http://dl.acm.org/citation.cfm?id=3026877.3026899

[2] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Data Engineering* 38, 4 (2015).

[3] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[4] Jeff Donahue. 2016. https://github.com/BVLC/caffe/tree/master/models/bvlc_reference_caffenet. (2016). [Online; Accessed 08-April-2018].

[5] Erik Erlandson. n.d.. http://erikerlandson.github.io/blog/2014/09/11/faster-random-samples-with-gap-sampling/. (n.d.). [Online; Accessed 23-January-2018].

[6] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: distributed graph-parallel computation on natural graphs.. In *OSDI*, Vol. 12. 2.

[7] Sumin Hong, Woohyuk Choi, and Won-Ki Jeong. 2017. GPU in-memory processing using Spark for iterative computation. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Press, 31–41.

[8] ImageNet. n.d.. http://www.image-net.org/. (n.d.). [Online; Accessed 01-August-2017].

[9] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, Vol. 41. ACM, 59–72.

[10] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 675–678.

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[12] Zhenyu Li, James Davis, and Stephen Jarvis. 2017. An Efficient Task-based All-Reduce for Machine Learning Applications. In *Proceedings of the Machine Learning on HPC Environments (MLHPC'17)*. ACM, New York, NY, USA, Article 2, 8 pages. https://doi.org/10.1145/3146347.3146350

[13] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment* 5, 8 (2012), 716–727.

[14] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 135–146.

[15] Lester Martin. 2016. https://martin.atlassian.net/wiki/spaces/lestermartin/blog/2016/05/19/67043332/why+spark+s+mapPartitions+transformation+is+faster+than+map+calls+your+function+once+partition+not+once+element. (2016). [Online; Accessed 08-April-2018].

[16] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I. Jordan. 2015. SparkNet: Training Deep Networks in Spark. *CoRR* abs/1511.06051 (2015).

[17] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 439–455.

[18] n.a. n.d.. http://www.deeplearning.net/software/theano/. (n.d.). [Online; Accessed 23-January-2018].

[19] n.a. n.d.. https://stackoverflow.com/questions/21185092/apache-spark-map-vs-mappartitions. (n.d.). [Online; Accessed 08-April-2018].

[20] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 147–156.

[21] Yuan Yuan, Meisam Fathi Salmi, Yin Huai, Kaibo Wang, Rubao Lee, and Xiaodong Zhang. 2016. Spark-GPU: An accelerated in-memory data processing engine on clusters. In *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 273–283.

[22] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301