École Polytechnique Fédérale de Lausanne

Master Thesis

Sparse Linear Algebra in the Deeplearning4j Framework

Author
Audrey Löffel
audrey.loeffel@epfl.ch

Supervisors
Prof. Martin Jäggi
MLO | EPFL
martin.jaggi@epfl.ch

François Garillot
Skymind Inc.
francois@skymind.io

August 17, 2017

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

skymind

Abstract

Deeplearning4j relies on Nd4j for storing off-heap and manipulating multi-dimensional arrays and for performing accelerated linear algebra operations. In this work, we present the study and implementation of different sparse storage formats such as the Coordinates format (COO) and Compressed Row format (CSR) into Nd4j. Then we explain the integration of the sparse BLAS routines of the Intel MKL library and the implementation of the native operations in Libnd4j. Finally, we discuss the optimizations that could be possible about the sparse formats or the library architecture.

Contents

1 Introduction

Linear algebra forms the backbone of many machine learning algorithms, especially in deep learning algorithms. It provides structures such as vectors and matrices to hold the data and parameters to perform operations between them.

Data is said sparse when it contains only a few non-zero values. That kind of dataset is really common in Machine Learning applications and can have a high influence on the computation.

Sparse Data is very common in Machine Learning

Many of machine learning applications tend to deal with a sparse data set. An example is the natural language processing applications, the words need to be vectorized before to be used by an ML application. The most common manner to achieve it is to use the bag of world model which uses the frequency of the words in the text. As stated by the Zipf' law [1], the frequency of any word in inversely proportional to its rank in the frequency table. It results in a very sparse data set.

Given the possible optimization that sparse data set allows and the high number of people that could take advantages of it, it becomes important to add the support of sparse data in the DL4J suite of libraries.

Moreover, nowadays deep-learning is a highly competitive field. Many libraries exist or are being built in many languages. However, Deeplearning4j has a prominent position in this ML-library war since it runs on the Java Virtual Machine (JVM) and business and production-oriented.

The Advantages of Sparse Data

Linear algebra is a branch of mathematics that is widely used throughout science and engineering. Machine learning and more specifically deep learning algorithms rely on linear algebra. Sparsity is a very useful property in Machine Learning. Some algorithms can have fast optimization, fast evaluation of the model, statistical robustness or other computational advantages.

A Real Case of Sparse Dataset

In 2008 Netflix launched a contest, the Netflix Grand Prize [2], to improve their recommender system model and to increase the accuracy of predictions. They published a sample data set made with the ratings of anonymous Netflix customers. The dataset had more than 100 millions sampled ratings and it contained about $m = 480'186$ users and $m = 17'770$ movies [3]. If stored as a dense matrix, it would need to store $8'532'905'220$ values in memory. That corresponds to a sparsity $\cong \frac{100'000'000}{8'532'905'220} = 0.011719338$.

Storing more than 8 billion 64-bit floating-point numbers needs more than 64 gigabytes of memory which quickly become unmanageable even for the world's fastest supercomputers.

Solution: Sparse Representation and Operations

Even if the compression of sparse data is very efficient when storing as dense data, we could gain in performance during the processing by using a sparse representation with sparse operations.

In sparse operations, only the values are accessed and processed while in dense operations, every cell is accessed even if it contains a zero value.

For example with a dataset with a density equal to 10% we only need 10% of the memory space and to process 10% of the matrix cells to perform an operation.

To reduce the size of the memory needed and to gain in performance, we must store the data in a sparse format. There are different kind of formats which each of them is more suitable for different aspects (Storage vs computation).

2 Sparsity and Formats

Definition

The sparsity of a dataset is defined by :

$$sparsity = \frac{\text{\# non-zero values}}{\text{\# values}} \tag{2.1}$$

Conversely when a dataset has only a few null values, the data are said dense. The density of the dataset is defined by the inverse of the sparsity:

$$density = \frac{1}{\text{sparsity}} \tag{2.2}$$

Formats

There are many methods for storing sparse data, each of them presents different advantages and disadvantages.

Matrices

Coordinates Format

It is the simplest method to encode a sparse array. The coordinates and the value of each non-zero entry are stored in arrays. Typically each element is encoded in a tuple (row, column, value)

This format provides an easy and fast way to retrieve a value and to insert a new non-zero element. It's also fast and simple to convert into a dense format.

$$A_{(M\times N)} = \begin{bmatrix} 0 & 2 & 0 \\ 0 & 0 & 3 \\ 1 & 0 & 4 \\ 0 & 0 & 0 \end{bmatrix} \quad \rightarrow \quad \begin{aligned} Values_{(1\times NNZ)} &= \begin{bmatrix} 2 & 3 & 1 & 4 \end{bmatrix} \\ Rows_{(1\times NNZ)} &= \begin{bmatrix} 0 & 1 & 2 & 2 \end{bmatrix} \\ Columns_{(1\times NNZ)} &= \begin{bmatrix} 1 & 2 & 0 & 2 \end{bmatrix} \end{aligned}$$

Figure 2.1: A matrix stored in COO format

But this format is not the most efficient regarding the memory consumption.

Compressed Row Format

The Compressed Row and the Compressed Column formats are the most general format to store a sparse array. They don't store any unnecessary element conversely to the COO format. But it requires more steps to access a element than the COO format.

Similarly to the row-major ordering explained in section 4.1, this format stores the values by row and use pointers to differentiate each row.

Each non-zero element of a row are stored contiguously in the memory. Each row are also contiguously stored.

The format, described by the Intel MKL Sparse Library [4], requires four arrays:

Values	All the nonzero values are store contiguously in an array. The array size is NNZ.
Column pointers	This array keeps the column position for each values.
Beginning of row pointers	Each pointer i points to the first element of the row i in the values array. The array size is the number of rows of the array.
End of row pointers	Each pointer i points to the first element in the values array that does not belong to the row i . The array size is the number of rows of the array.

Compressed Column Format

The Compressed Column Format is similar to CSR but it compresses columns instead of rows. The values are stored in a column-major ordering, as explained in section 4.1.

Given a matrix $N \times M$, the pointers arrays will have a size M.

$$A_{(N\times M)} = \begin{bmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 4 & 0 \\ 0 & 0 & 2 & 1 \end{bmatrix} \rightarrow \begin{aligned} Values_{(1\times NNZ)} &= \begin{bmatrix} 2 & 3 & 1 & 4 & 2 & 1 \end{bmatrix} \\ Columns_{(1\times NNZ)} &= \begin{bmatrix} 1 & 2 & 0 & 2 & 2 & 3 \end{bmatrix} \\ pointersB_{(1\times N)} &= \begin{bmatrix} 0 & 1 & 2 & 2 & 4 \end{bmatrix} \\ PointersE_{(1\times N)} &= \begin{bmatrix} 1 & 2 & 2 & 4 & 6 \end{bmatrix} \end{aligned}$$

Figure 2.2: A matrix stored in CSR format

Tensors - Multi-dimensional arrays

A tensor is a multi-dimensional array. The order of the tensor is its number of dimension. It corresponds to the number of indexes needed to index a value. Matrices, vectors and scalars can be represented as tensors where the order is equals to 2, 1 and 0 respectively.

This generalization allows a more generic implementation of a n-dimensional array in the Nd4j library.

Coordinates Format

The COO format can easily be extended to encode tensors by storing an array of indexes instead the row and column coordinates.

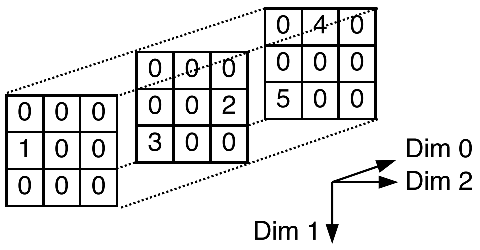


Figure 2.3: A 3 × 3 × 3 tensor

The tensor, illustrated in figure 2.3, of order $K = 3$ with shape $3 \times 3 \times 3$ which has the following non-zero values :

value	indexes
1	[0, 1, 0]
2	[1, 1, 2]
3	[1, 2, 0]
4	[2, 0, 1]
5	[2, 2, 0]

It can be encoded with one values array and one indexes array :

$$Values_{(1\times NNZ)} = \begin{bmatrix}1, & 2, & 3, & 4, & 5\end{bmatrix}$$
$$Indexes_{(NNZ\times K)} = \begin{bmatrix}[0,1,0], & [1,1,2], & [1,2,0], & [2,0,1], & [2,2,0]\end{bmatrix}$$

Figure 2.4: A tensor stored in COO format

3 The Deeplearning4j Library

Deeplearning4j is an open-source, distributed, deep learning library for Java and Scala. Integrated with Hadoop and Spark, DL4J is specifically designed to run in business environments on distributed GPUs and CPUs.

Architecture of the library

The library is composed by several sub-libraries:

Deeplearning4j	provides the tools to implement neural networks and build computation graphs
Nd4j	is the mathematical back-end of Deeplearning4j. It provides the data structures for the n-dimensional arrays and allow Java to access the native libraries via JavaCPP and the Java Native Interface.
Libnd4j	is the computing library that provides native operations on CPU and GPU. It's written in C++ and Cuda.
Datavec	provides the operations for the data processing such that data ingestion, normalization and transformation into feature vectors.

The Importance of Nd4j in the Library

Nd4j (N-Dimensional Arrays for Java) is at the base of the Deeplearning4j library, it provides data storage, manipulations, and operations. It gives the atomic pieces needed to build more complex deep learning systems. It is a scientific computing library for the JVM. It exposes n-dimensional arrays that can be used in linear algebra and large-scale matrix manipulation. It supports CPU and GPU computations through interchangeable backends. The different backends use the same interface: they can be swapped without any change to the implementation code.

The APIs provided by the library are essentially wrappers for the different version of BLAS (Basic Linear Algebra Subprogram).

BLAS is a specification that defines the low-level routines for linear algebra operations (for vectors and matrices). There exist several libraries implementing those subroutines in C or Fortran for dense or sparse formats. In Nd4j the BLAS subroutines can directly be called from Java thanks to JavaCPP, that internally uses the Java Native Interface (JNI) to call native routines from the JVM environment. This architecture allows the library to benefit from the advantages of the native side.

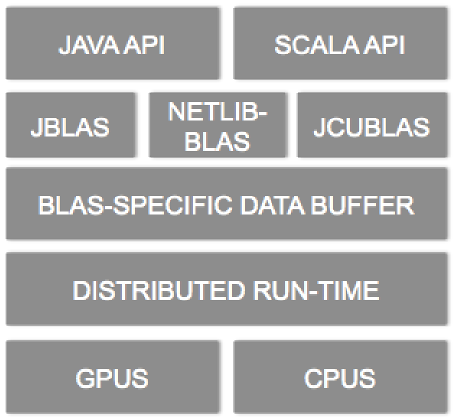


Figure 3.1: Nd4j architecture

Nd4j needs a Sparse Representation

In Deeplearning4j, Sparse Data is treated as dense and use the dense operations of BLAS and Libnd4j to perform computations. With a new sparse representation, we can gain in storage space and computation speed.

Currently, there is no library in the JVM ecosystem that provides the sparse representations and the accelerated operations. For example, Breeze library [5] does support the sparse vectors and the sparse CSR matrices but does not have sparse tensor. Moreover, the operations provided by Breeze are not accelerated.

Nd4j will be the first JVM library to support the sparse tensors with their accelerated operations.

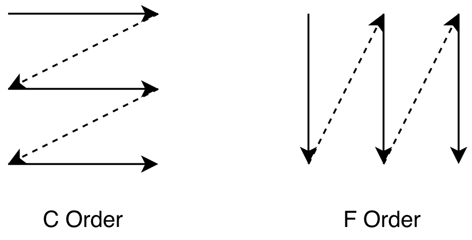
4 Structure of an Multi-dimensional Array

The new sparse array have to be compliant with the API and inter-operable with the current dense array implementation.

Storing an Array

A dense array is stored as a single contiguous block of memory, flattened in a one-dimensional array. Arrays are stored off-heap (outside the JVM memory management). The reasons behind this design decision are numerous: better performance (avoid the interruptions of the garbage collector), better interoperability with BLAS libraries, and to avoid the disadvantages of the JVM such as the limited size of arrays due to the integer indexing (limited to $2^{31}-1 \cong 2.14$ billion elements)

There are two methods to store a multi-dimensional array into a linear memory space: row-major order (C) or column-major order (Fortran). Figure 4.1 shows how a two-dimensional array is stored according to the order.



$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix} \Rightarrow \begin{matrix} C-Order = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{21} & a_{22} & a_{23} \end{bmatrix} \\ F-Order = \begin{bmatrix} a_{11} & a_{21} & a_{12} & a_{22} & a_{31} & a_{13} \end{bmatrix} \end{matrix}$$

Figure 4.1: Comparison between C-order and F-order

The data is accessed via strides which define how to index a contiguous block of data. For each dimension, it defines by how many values two consecutive elements are separated. In the case of matrix A defined in figure 4.1, the strides would be $(3,1)$ in case of C-order and $(1,3)$ in case of F-order. Strides $(3,1)$ means that each row is separated by 3 values and each column is separated by 1 value.

Data Buffer

The *DataBuffer* is a storage abstraction which provides optimal storage and retrieval depending on the backend. The data is stored off-heap through JavaCPP. It is basically a wrapper around a pointer and an indexer with utility methods to access and modify the data. The pointer points to the allocated memory space and the indexes provides an easy-to-use and efficient way to access a multi-dimensional memory space.

The implementation of the *DataBuffer* depends on the data type, because of the length needed to store a single value (int -> 32 bits, long -> 64bits, float -> 32bits, double -> 64bits, assuming a 64bit-architecture).

In the GPU backend, the data is allocated on the host (in the RAM) and on the device (on the GPU), the *DataBuffer* has two pointers instead of one.

Parameters of an Array

The information about the shape of the array are grouped in a *DataBuffer* object called ShapeInformation. It groups these following information:

Rank	The number of dimension of the array.
Shape	The shape of the array.
Strides	Provides information about the logical layout of the array for each dimension.
Offset	Provides the position of the first value in memory that belongs to the array or view (explained in section 4.2).
ElementWiseStride	Indicates how two contiguous elements are physically separated in memory.
Order	C or F order

Views

The data in memory can be shared by multiple *NDArrays*. An *NDArray* can refer to a subset of another *NDArray*. We say that such an array is a view of the original array. This is a powerful concept that avoid the unnecessary copy of the data which is a very expensive operation.

Since the memory space is shared, changes to array will impact the content of the other arrays. Figure 4.2 illustrates how a view shares its data with its original array.

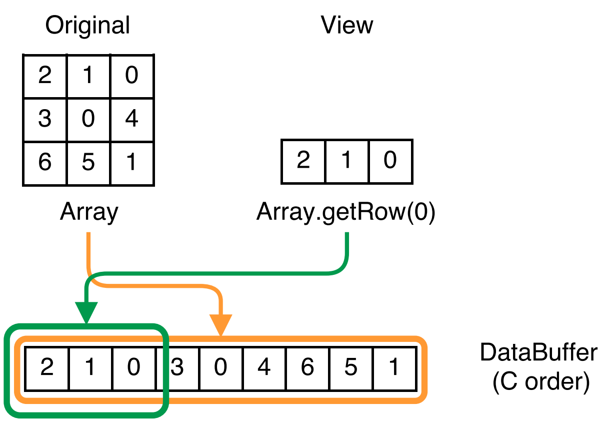


Figure 4.2: View shares memory with the original array

We can also get a part of a view. In this case we have a view of a view and the all share the memory space of the original array. The figure 4.3 shows a possible hierarchy of views. The DataBuffer of the views contains a pointer to the original DataBuffer and a pointer the memory space.

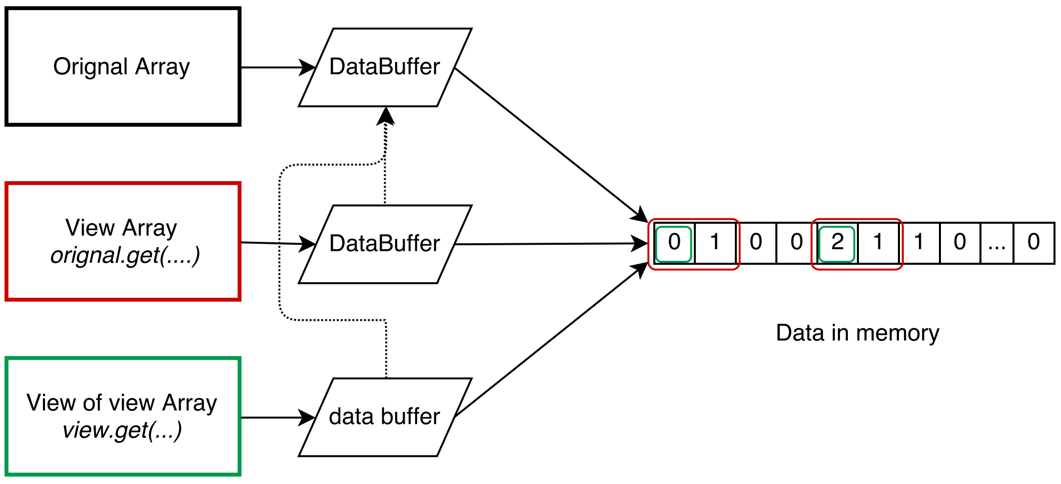


Figure 4.3: View hierarchy

Indexes

NDArrays can be accessed through a combination of indexes. There are many way to index an array. For each dimension of the original array, we can specify if we want everything, an interval or a specific value of it. It gives a lot of flexibility to access sub-part of a array.

Nd4j emulates the syntax of Numpy[6] and use a similar mechanism to indexes the arrays. Among the different indexes we can note the intervals, the points,. . .

Each index implements the interface *INDArrayIndex* and extends from *NDArrayIndex*.

The indexes provide an efficient mechanism to access Tensors along dimension. A tensor along dimension (TAD) is a view of an actual tensor with a lower rank. The TAD are a very powerful concept to perform fast and optimized operations on tensors.

Operations

Nd4j defines five types of operations:

1. Scalar: element-wise operations such as addition or multiplication where the same operation is applied on each element of the array.
2. Transform: in-place operations such as logarithm or cosine. There are usually executed in an element-wise manner but it's not always the case; some can be applied along a dimension.
3. Accumulation: Also called Reduction. Operations applied over the whole ndarray or along a dimension such as the sum of all values.
4. Index Accumulation: Similar to accumulation operations but return an index instead of a value. A classic operation is getting the maximum or minimum value along a dimension.
5. Broadcast: Some of the more useful operations are vector operations, such as *addRowVector* which add a row vector to every row of a matrix.

5 Implementation

Hierarchy of Arrays

The API of an array is defined by an interface called *INDArray* which has a dense implementation for each backend: *NDArray* class for the CPU and *JCublasNDArray* class for the GPU. But since most of the operations and methods are shared between the two dense backends, they are implemented in an abstract class called *BaseNDArray*.

Adding sparse representations asked two questions:

1. What can be shared with the dense arrays ?
2. What can be shared between the different sparse arrays and what is format-specific ?

To answer those questions, we need to go a little bit deeper in the implementation.

The dense implementation includes methods that are inherently related to the way dense arrays are internally made, and other methods are related to the generic parameters such as the shape or are utility methods, and can be used independently of the internal implementation.

The first type of methods is not useful in case of sparse. Dense and sparse arrays are not built in the same manner. Methods such as *getStrides* are not relevant in the sparse context. Reciprocally the sparse array will need methods which will be irrelevant in the dense context.

We encounter the same situation between the different sparse formats. Some will need utility methods that the other ones won't need.

However some methods should be reusable such as traversal operations which operate in an element-wise manner.

But as much as possible has to be exposed in the *INDArray* interface. To avoid code duplication, everything than can be shared should be implemented in the higher level of the hierarchy.

The methods that are not compatible with a type of array will simply throw an unsupported operation exception.

The drawback brought with this solution is that we always need to verify the type of the array before doing any operations.

Figure 5.1: Arrays hierarchy in Nd4j

Limitations and Constraints

Nd4j has been made with the perspective of dense arrays. The design has been thought and optimized regarding the dense implementation which brings some constraints to implement the sparse representation

Storing off-heap

The data, encapsulated into DataBuffer object, is stored off-heap. Several reasons drive this design decision.

1. The size of the memory that can be allocated within the JVM is limited to 2 billion. We can not fit some big matrices into the JVM.
2. We want to use Cuda to accelerate the computation on the GPU. Cuda memory managment is not compatible with the JVM environment, the data has to be stored on the native side.

   In Cuda backend we store the data in a dual manner: one host copy (in the RAM) and one device copy on (on the GPU).
3. Java execution is always sequential even with multi-threading, it cannot be parallelized on multiple CPUs. With C++ we can have cheaper parallel executions.
4. At C++ level we can have easier access to SIMD (Single Instruction, Multiple Data) instructions that allow to execute an instruction in parallel on different blocks of the same data. Those instructions are heavily exploited by BLAS.

Workspaces

The garbage collector makes the memory management of the JVM easier. It takes care of tracking the referenced object created by the program and remove them if there is no any longer reference pointing to the object. It automatically frees the unused allocated memory space. But the free operation is not immediate. The garbage collector has to pause the execution of the program at some point to modify the memory. Some unused memory spaces can still be temporarily allocated.

In machine learning and deep learning, many of the operations have a cyclic workload: we get a part of a dataset, we perform an operation and then we pass to the next data. Only the current values a relevant, the old memory content is not used anymore. We want to avoid having too much previous data allocated.

Nd4j has implemented workspaces. A workspace is a memory chunk that we allocate once and which we can reused it as long as we need. The garbage collect does not manage this memory space. In a cyclic workload, we do not allocated a new memory space, we can use the space used by the last operation. The memory space size do not change, we only replace the content.

DataBuffers have a fixed length

Once allocated a *DataBuffer* can not be reallocated. It is not a issue when manipulating dense arrays, we never need to add a new value in a dense array, we only update the existing values. However we need to be able to add new values in the sparse buffers. Each time a zero value become a non-zero value we have to put the value and its indexes in memory.

To reallocate a buffer we need to:

1. Create a new *Pointer* to a bigger memory space and replace the current pointer of the *DataBuffer* .
2. Create a new *Indexer* with the new *Pointer* and replace the current indexer of the *DataBuffer*.
3. Copy the content from the old *Pointer* to the new space.
4. The buffer has two length attributes: *length* which count the number of values in the buffer and *underlyingLength* which reflects the actual allocated size in memory. After the reallocation the *underlyingLength* is increased to the new size, but the *length* only increases at the insertion of a new value.

We over-allocate the buffer in order to limit the amount of reallocation operations. Instead to expand the size to the new data length, we double the available memory space. The next time we want to insert some new values in the buffer, we avoid the expensive costs of the reallocation and the data copy.

Another possibility would have been to create a new *DataBuffer* with the content of the old buffer plus the new value. But changing the DataBuffer object in a array would have broken the view mechanism since the views would still point to the old *DataBuffer* reference. Moreover this solution doe not allow the over-allocation.

CSR Matrices Implementation

Nd4j does not provide any representation for this matrix format. We used the existing data structure, parameters holder and indexes to implement a new representation and the operations.

Structure

The representation uses 4 data buffers to encode a CSR matrix. One for the non-zero values, one for the columns indexes and two for the row pointers (to the beginning and to the end of each row).

Put a value

To insert a new value or to update an existing non-zero value, we need to identify where the values of row we want ot insert to are located in the values buffer and in the columns buffer. The beginning and end of rows pointers give us the range of indexes.

While iterating over the range of values, if we find a value with the same column index than the one we want to insert to, we can update the values and nothing needs to be changed in the three other buffers.

However if there is no value with that column, we need to insert a new one at the correct position. Then we need to update the end of row pointer for this row. Finally each row pointers that come after need to be increment by one.

The implementation of this method is shown in appendix A.2.

Get a Sub-array

1. First, we need to compute the parameters such as the shape, the offsets,. . . of the sub-array. We use the *ShapeOffsetResolution* which resolves a combination of indexes and returns the parameters. Despite this class has been made for resolving the indexes of a dense representation, we can reuse it in case of sparse array. The parameters we need are computed independently of the storage format and the internal structure of the array.

   For example an interval $[1,3[$ on a dimension i would result to $offsets[i] = 1$ and $shape[i] = 2$, which is the length of the interval. It does not make any assumption about the underlying storage format.

   Here are the parameters we need to compute:

   - the shape: an array with two elements containing the new shape of the sub-array.
   - the offsets: an array with two elements that indicate the first row and the first

column that belongs to the sub-array.

- the offset: indicates the position in the data buffer of the first element that belongs to the sub-array.

2. Sometimes the offsets are equal to zeros while having a non-zero offset. This situation occurs for example when using only a combination of *all* and *point* indexes. In this case we need to compute the offsets manually.

Algorithm 1 Calculate the offsets

```
procedure OFFSETS(int $offset$)
  ▷ the shape array mentioned behind is the shape of the current array from which we are getting a view.
    $offsets \leftarrow new\ int[2]$
    $offsets[0] \leftarrow \lfloor offset\ /\ shape[1] \rfloor$
    $offsets[1] \leftarrow offset\ \%\ shape[1]$
end procedure
```

3. With the offsets and the shape we can define the bounds of each dimensions such has $dim_i \in [offsets[i], offsets[i] + shape[i][$

4. Finally we need to loop over every element of the original array. We have to reconstruct the pointers buffers step-by-step by checking if the coordinates of the current element are included in the bounds and update the pointers buffers if needed.

The entire method is shown in appendix A.1.

Limits of this format

This format only works with two dimensions and cannot be extended to tensors. Therefore it makes it hardly compatible with the API. Moreover the operations to get or put values aren't straightforward. Several step are necessary before accessing the value of a given coordinates:

Assuming we want to get the value v_i of the coordinates (r_i, c_i). We have to get the two row pointers of r_i, iterate over the values buffer on the range delimited by the pointers and check if the column index of the current value is equal to the c_i. As soon as we match the column index, we can return the value v_i or we return zero if we reach the end of the row without matching.

However we could use this format to represent Tensors Along Dimension (TADs) as explained in section 8.4

COO Tensors

Conversely to the CSR format, the COO format can easily be extended to multi-dimensional array. Since we stored the coordinates of each value, the rank of the array is not limited to two, we can store any number of coordinates.

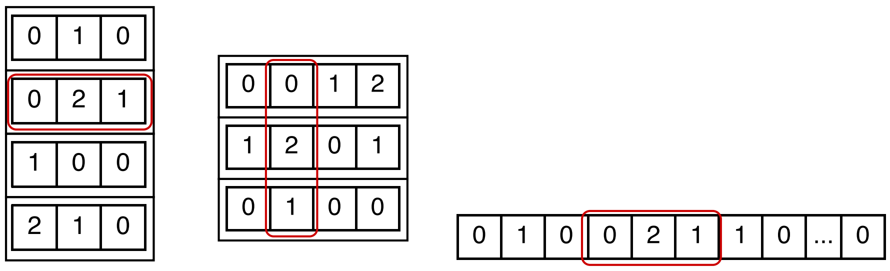
As the CSR representation, no implementation of a COO format exists in Nd4j, we have to create it using the existing components of the library.

Naive implementation

Based on the description in 2.2.2 the COO encoding needs one data buffer to store all the non-null values and one for the indexes of each values.

An easy solution would have been to store the indexes into a multi-dimension array of *DataBuffers*: One buffer for each value, or one buffer for each dimension. Due to the native constraints that makes hardly manageable to have such arrays (difficult to pass the array to the native side and Cuda side), we choose to flatten the indexes into one buffer.

The figure 5.2 illustrates the different manner to store the coordinates $[0,2,1]$ of a value v_i in a 3-order tensor with four non-zero values.



(a) each index is stored contiguously (b) Each dimension is stored contiguously (c) Flattened Indexes

Figure 5.2: Illustration of the different possible data structure for storing the indexes [0, 2, 1] of a value v.

But this implementation makes difficult to be compatible with the API. It brings several issues:

- The key of views is the sharing of their data to avoid the reallocation. In case of COO format, views have to share the data buffer and the indexes buffer. We might be tempted to create an new indexes buffer for each view. Each indexes buffers will have the coordinates of the value regarding the view context. But it would not be possible to add a value in the original array by adding it in a view. If we only put the new value in the shared value buffer without updating the indexes, the original array would have a value buffer bigger than its indexes buffer and we would not know what the indexes of this value are.

Even when sharing both buffers, how would we know which value is included in the view and which is not?

- The coordinates of a value in a view are not necessary the sames of the same value in the original array.

  They can be offset if dimension is partially included in the view. Figure 5.3 shows a matrix an a view (in red). The value$v_i = 5$ would have the coordinates [1, 1] in the original array while it has the coordinates [0, 0] in the view. How can the indexes be translated between views?

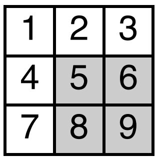


Figure 5.3: A 3 × 3 matrix with a 2 × 2 view in grey

- A view can have a lower or higher rank than its original array. How can the view indexes be stored if they do not have the same dimensionality?

Reverse coordinates

An important and frequently used method of the API is to get the value of a set of coordinates. Since the indexes are flattened into one buffer, we need to find them into the buffer and get the underlying index and then the corresponding value.

Since the indexes buffer is lexicographically sorted, a binary search can be applied. But instead of comparing a value, we need to compare a sub-array of the buffer with the target indexes. An additional step is necessary before the comparison: translate the position of the value in the values buffer to get the positions of the indexes in the buffer.

We iterate over the values (NNZ elements) and for each value v_i, we extract the sub-array that contains the coordinates of v_i.

The subarray of a value v_i is contained between the following indexes:

```
idx_from = underlyingRank * i;
idx_end = idx_from + underlyingRank;
```

The underlyingRank is the number of coordinates used to index a single value in the original array buffer.

Put a Value

Several steps are needed to put a new value in a COO array:

1. The same value in a view or in the original array can have different coordinates. A translation step is needed between the view indexes and the original array coordinates (the actual coordinates stored in the shared buffer).
2. Verify if the old value of this index is already non-zero. If so we can either remove the entry if the new value is zero, or replace the old value by the new in the buffer. Nothing else need to be done.
3. If the new value is equal to 0: the indexes and the value are not added in the buffers.
4. If it is a new non-zero value: we need to insert the value and the indexes in the buffers. But first we need to ensure that the buffers have enough spaces for the new entries. If not, we need to reallocate them. The new entries are added at the end of the buffers; the sort are not maintained at insertion to avoid sorting the array too often when it is not needed.

   For example an operation that modifies sequentially the values (i.e traversal operation) might need to add multiple non-zero values in memory. If the sort was maintained at insertion, we would need to pay the cost of searching the right position for each value.

More parameters are needed to define the tensors

Nd4j uses a powerful tool to access part of an array: the *INDArrayIndexes*. But they are the sources of many of the issues cited in section 5.4.1.

We present each type of index, with their set of constraints and the solutions implemented below.

Resolution of the Indexes

The *ShapeOffsetResolution* is a key mechanism in the resolution of the indexes. Taking a combination in input, it computes the information such as the shape, the offsets, the strides,... about the sub-array selected by the indexes.

Despite this class has been made for resolving the indexes of a dense representation, we can reuse with sparse arrays. The parameters we need are computed independently of the storage format and the internal structure of the array.

For example an interval [1, 3[on a dimension i would result to $offsets[i] = 1$ and $shape[i] = 2$, which is the length of the interval. It does not make any assumption about the underlying storage format.

All Indexes

All indexes are the most straightforward of the library. They are used to collect all the elements of a dimension. They are useful when used in combination with other type of indexes. For example if we would like to take the first column of a matrix, we would need to ask for the first element of all the rows.

Interval Indexes

Interval indexes takes an contiguous subpart of a dimension containing in an interval. They don't modify the rank

The grey sub-array in figure 5.4 is the result of the operation :

```
    myArray.get(NDArrayIndex.interval(1, 3), NDArrayIndex.interval(1, 3));
```

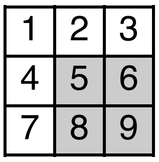


Figure 5.4: A 3 × 3 matrix with a 2 × 2 view in grey

After the resolution of the indexes, we obtain offsets equal to [1, 1] with an offset equal to 4, which mean that the first row and the first column are not included in the view and the first element in memory that belongs to the view is at position 4 in the original array (the value 5 in the figure).

To be able to identify what values belong to the view and what values do not, we need to define the bounds of each dimension. For this example, we have

$(idx_{row}, idx_{colum}) \in view$ if $idx_{row} \in [1, 3[$ and $idx_{column} \in [1, 3[$

We only need to store the lower bound i_{lower} of the interval $idx_{dim} \in [i_{lower}, i_{upper}[$, we can easily compute $i_{upper} = i_{lower} + shape[dim]$. The lower bounds are stored in the *sparseOffset* array.

Point Index

Point indexes take one unique element of a dimension. They reduce the rank of the array.

Assuming we have a tensor with a shape equal to (2 × 3 × 3) as shown in figure 5.5. We want to take a slice along the first dimension, assuming that the first dimension is called the pages, the second the rows and the third the columns. We use the following operation to get the first page:

```
    myTensor.get(NDArrayIndex.point(0),
        NDArrayIndex.all(),
        NDArrayIndex.all());
```

The result is a view with a shape equals to (3 × 3), as shown in grey in figure 5.5

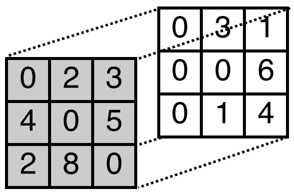


Figure 5.5: A 2 × 3 × 3 tensor with a slice along the first dimension in grey

The coordinates of the resulting matrix have only two dimensions instead of three. We have an issue when trying to access a value given a pair of coordinates in the view context. There is no direct matching between these coordinates and those who actually are in the indexes buffer.

In the example above, the indexes buffer is equal to $[0,0,1,0,0,2,0,1,0,\dots]$ which correspond to the values buffer $[2,3,4,\dots]$. The value 4 has the coordinates $(0,1,0)$ while its coordinates in the view are equal to $(0,1)$.

Generalizing, each value of the tensor has its coordinates in the form (idx_0, idx_1, idx_2). But in the resulting view the values are defined with only two coordinates: (idx_1, idx_2), the first dimension is fixed with a value equal to 0.

To translate the view coordinates to the original coordinates which are actually stored into the shared buffer, we need to keep track of the unused dimension and its value. The solution is to add an additional parameter array that keeps track of the status of each dimension. The array is called *flags* and it can contains either 0, which means *active*, or 1 which means *fixed*.

In this example the flags array would be equal to [1, 0, 0] because the first dimension is fixed at position 0. The value 0 would be keep into the *sparseOffset* array. And the indexes translation would be : $\mathrm{T}(idx_1, idx_2) = (sparseOffset[0], idx_1, idx_2)$

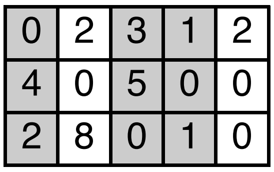
Specified Index

Specified index will take non-contiguous values of a dimension, depending on the position in the indexes array when calling $myArray.get(indexes...)$. This index takes an array as argument containing the values we want to select.

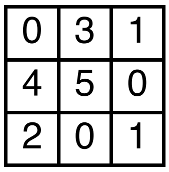
Assuming we have an 3 × 5 matrix, as shown in figure 5.6a, on which we call the following operation to select the first, the third and the fourth columns :

```
    myMatrix.get(NDArrayIndex.all(),
        new SpecifiedIndex(0, 2, 3));
```

Instead of returning a view of $myMatrix$, it returns a new array containing the data from the original matrix. The data is copied, not shared. The data of the view that would result from this operation is not easily indexable in memory with stride and offsets and it would be difficult to represent it. Therefore the figure 5.6b is an independent 3 × 3 matrix resulting from the above operation.



(a) A 3 × 5 matrix with selected columns in grey



(b) A 2×2 matrix created from the column of the matrix in figure 5.6a

Figure 5.6: Example of a specified index behavior on a matrix

New Axis Index

New axis indexes are used to add a new dimension to the array. The new dimension always has a length equal to 1. It can be prepended or inserted in the middle of the dimensions. Since the rank is higher, more coordinates are needed to access a value. However the shared indexes buffer is limited to the original rank. Similarly to the point index, we need a new parameter array that keep tracks of the position of the new dimensions to be able to translate the coordinates from view to original context.

Assuming we have a 3 × 3 matrix: calling *myArray.get(NDArrayIndex.newAxis(), NDArrayIndex.all())* will prepend a new dimension. The view is a 1 × 3 × 3 tensor with a hidden dimension parameter array equal to [0].

We use an array called *hiddenDimension* to keep track of the position of the new dimensions in the coordinates. In the example above, each value v_i has the coordinates (i_0, i_1, i_2) within the view context. The array *hiddenDimension* contains [0] because this dimension does not exist in the original array.

Sparse Indexes Translation

To translate the view coordinates to the underlying coordinates, we have to take into accounts: the sparse offsets, the hidden dimensions and the flags. With those three parameters we have everything needed for translating.

While iterating over the view dimensions, three situations can happen :

1. The current dimension is hidden; the original array does not contain it becaise it has

been created by a *point* index: We do nothing. The dimension is skipped and we start the next iteration.

2. The current dimension is fixed: We put the corresponding offset into the result array. Since several fixed dimensions can occur one after the others in the coordinates array, we need to repeat until we reach an active dimension.
3. The current dimension is active: We sum the view coordinate with the sparse offset.

This method is described in detail in algorithm 2

Algorithm 2 Translate the indexes from view to underlying context

```
procedure TRANSLATE(int[] $viewIndexes$)
        ▷ $underlyingRank$ is the rank of the original array which is equal to the number of indexes stored in the buffer for a given value. $hiddenDimensions$ contains the position of each hidden dimension. $sparseOffsets$ contains the offsets for each dimension. They are class fields of the COO array instance.

    $result \leftarrow new\ int[underlyingRank]$
    $idxPhy \leftarrow 0$

    $hidden \leftarrow 0$                              ▷ Number of hidden dimension encountered

    for $idxVir = 0$ to $viewIndexes.length$ do
                        ▷ The current dimension is hidden, it does not appear in the result
        if $hidden < hiddenDimension.length\ \&\ hiddenDimension[hidden] == idxVir$ then
            $hidden \leftarrow hidden + 1$
        else
            while $idxPhy < underlyingRank\ \&\ isFixed(idxPhy)$ do
                $result[idxPhy] \leftarrow sparseOffsets[idxPhy]$  ▷ If the dimension is fixed, the coordinate takes the value of the offset
                $idxPhy \leftarrow idxPhy + 1$
            end while
                        ▷ If the dimension is not fixed, we add the offset to the coordinate
            if $idxPhy < underlyingRank\ \&!isFixed(idxPhy)$ then
                $result[idxPhy] \leftarrow sparseOffsets[idxPhy] + viewIndexes[idxVir]$
                $idxPhy \leftarrow idxPhy + 1$
            end if
        end if
    end forreturn $result$                   ▷ contains the indexes of the underlying array
end procedure
```

Example of indexes translation

An example of execution is explained in appendix A.1.2.

Computations of the the Parameters

Computation of the Sparse Offsets

The *sparse offsets* are computed from the offset resulting from the resolution of the indexes. The *offset* gives us the position of the first element in the array.

1. For each dimension except the innermost one, we divide the offset by the length of that dimension. The quotient gives us the sparse offset for the dimension. Then we need to remove the number of elements that are in the same dimension but at a lowest value (if the element is in the 4th row, we want to remove all the elements of the previous rows).
2. We reached the last dimension: To compute the sparse offset of the last dimension we need to take the modulo of the updated offset by length of the dimension.

   In steps 1 and 2 there is only one operation to compute the sparse offset per dimension, then for a n-order tensor, the complexity of these steps is O(n).
3. We have the sparse offsets given a set of indexes. But if we are computing the sparse offsets of a view of a view, the view from which we are taking a view might already have some sparse offsets. We need to merge the new sparse offsets with those of the current array. The final result is the sum of the existing offset and the freshly computed offset.

   During the merge we should be particularly careful with fixed dimensions of the current array because they are absent from the sparse offset resolution explained above. The result does not contain any information about them, we need to add these fixed dimensions at the correct position with their current sparse offset to final offsets array.

Example of Sparse Offsets Computation

An example of the execution of the algorithm is presented in appendix A.1.1.

Computation of the Flags

The *Flags* array determines which dimension is active and which one is hidden from the point of view of the array. The dimension can only be reduced using a *point* index, which means the flags can be computed during the offset resolution. An *interval* with a length equals 1 does not reduce the dimensionality of the array. We fill the flags array while iterating over the indexes array.

Computation of the Hidden Dimensions

The resolution returns an array containing the position of the *newAxis* indexes in the indexes array. The array may already have some hidden dimensions. In this case we need to adapt the

Algorithm 3 Calculate the sparse offsets

```
procedure CreateSparseOffsets(int $offset$)
▷ $rank$, $underlyingRank$ (the rank of the original array and the number of dimension of the indexes buffer), $shape$ and $sparseOffsets$ are class fields of the array instance

    $newOffsets \leftarrow new\ int[rank]$                    ▷ Compute the new offsets
    for $i = 0$ to $(rank - 2)$ do
        $nbElements \leftarrow \prod_{j=i+1}^{rank} shape[j]$
        $newOffsets[i] \leftarrow \lfloor offset \div nbElements \rfloor$
        $offset \leftarrow offset - newOffsets[i] \times nbElements$
    end for
    $newOffsets[rank - 1] \leftarrow offset \mod shape[rank - 1]$


    $finalOffsets \leftarrow new\ int[underlyingRank]$ ▷ Merge with sparseOffsets of this array
    $active \leftarrow 0$
    for $i = 0$ to $underlyingRanke$ do
        if $flags[i] == 0$ then        ▷ If the dimension is already inactive in the current array
            $finalOffsets[i] \leftarrow sparseOffsets[i]$
        else
            $finalOffsets[i] \leftarrow newOffsets[active] + sparseOffsets[i]$
            $i \leftarrow i + 1$
        end if
    end forreturn $finalOffsets$
end procedure
```

result with the current hidden dimension.

Algorithm 4 Calculate the hidden dimensions

```
procedure CreateHiddenDimensions(int[] $newAxis$)
                                        ▷ $hiddenDimensions$ is a class field of the array

    if $newAxis$ is empty or null then return $hiddenDimensions$
    end if

    if $hiddenDimensions$ is empty then return $newAxis$
    end if

    $size \leftarrow newAxis.length + hiddenDimensions.length$      ▷ Merge both arrays
    $newArray \leftarrow$ new $int[size]$, $newArrayIdx \leftarrow 0$, $newDim \leftarrow 0$

    for ($oldDim = 0$ to $hiddenDimensions.length$) do
        while (($oldDim \geq hiddenDimensions.length \parallel newAxis[newDim] \leq hiddenDimensions[i]$) && $newDim < newAxis.length$) do
            $newArray[i] \leftarrow newAxis[newDim] + oldDim$
            $newArrayIdx \leftarrow newArrayIdx + 1$
            $newDim \leftarrow newDim + 1$
        end while
        $newArray[i] \leftarrow hiddenDimensions[oldDim] + newDim$
        $newArrayIdx \leftarrow newArrayIdx + 1$
    end forreturn $newArray$
end procedure
```

Final Implementation

The final representation is encoded with 5 *DataBuffers*:

1. Values: Store the values linearly.
2. Indexes: Store the flattened indexes of each values.
3. Flags: Define which dimensions are active and which are fixed.
4. Sparse Offsets: Define the bounds of each dimension.
5. Hidden Dimensions: Keep track of the position of the hidden dimension in the shape array .

The indexes and values are sorted in the lexicographic order in order to make the search by indexes easily via binary search. We sort the buffers only before we need to read them : when performing an operation or when accessing its contents. The sort is not maintained at the insertion.

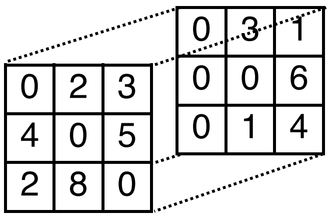
The flags, the offsets and the hidden dimensions arrays are grouped in one *DataBuffer*, similarly to the *ShapeInformation* buffer.

Example

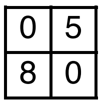
Figure 5.7a shows a tensor T with a shape $(2 \times 3 \times 3)$ and Figure 5.7b shows the view resulting from the operation below:

```
T_view = T.get(NDArrayIndex.newAxis(),
    NDArrayIndex.point(0),
    NDArrayIndex.interval(1, 3),
    NDArrayIndex.interval(1, 3));
```

The first index prepends a new dimension and increases the rank, the second takes the first page of the tensor and reduces the rank. Then the next ones select the sub-array in the first page dimension. The parameters of T_{view} are listed in figure 5.8



(a) A tensor T with shape $(2\times3\times3)$



(b) A view of tensor T with shape $(1 \times 2 \times 2)$

Figure 5.7: A tensor and a view of it

$$Shape = \begin{bmatrix}1 & 2 & 2\end{bmatrix} \quad Values = \begin{bmatrix}5 & 8\end{bmatrix}$$
$$Indexes = \begin{bmatrix}0 & 1 & 2 & 0 & 2 & 1\end{bmatrix} \quad Flags = \begin{bmatrix}1 & 0 & 0\end{bmatrix}$$
$$SparseOffsets = \begin{bmatrix}0 & 1 & 1\end{bmatrix} \quad HiddenDimension = \begin{bmatrix}0\end{bmatrix}$$

$$Values = \begin{bmatrix}2 & 3 & 4 & 5 & 2 & 8 & 3 & \dots & 4\end{bmatrix}$$
$$Indexes = \begin{bmatrix}0 & 0 & 1 & 0 & 0 & 2 & 0 & \dots & 2\end{bmatrix}$$

Figure 5.8: Parameters defining the view T_{view} in 5.7b

Get a Sub-Array

The new parameters make the task easier. Once the indexes resolved and the parameters computed, a new array is created with the current values and indexes buffers and the new parameters.

However if one of the indexes is a *SpecifiedIndex* we need to create a new array and copy the data. In this case the index resolution translates any type of indexes to a specified index:

Assuming we have a $3 \times 3 \times 3 \times 3$ 4-order tensor. The indexes array of the following operation:

```
myTensor.get(NDArrayIndex.all(),
    NDArrayIndex.point(1),
    NDArrayIndex.interval(0, 2),
    new SpecifiedIndex(0,2));
```

would become : [SpecifiedIndex(0,1,2), SpecifiedIndex(1), SpecifiedIndex(0,1), SpecifiedIndex(0,2)].

Then we iterate over all the included combinations of coordinates and add the elements into a new empty array.

6 Operations

Basic Linear Algebra Subprograms (BLAS)

As introduced in section 3.2, BLAS defines the low-level routines for linear algebra operations (for vectors and matrices). Nd4j supports OpenBlas and Intel MKL as the CPU's libraries and Cuda (Cublas) as the GPU's library.

Those libraries are using a dense format only. For sparse computation we need to support two additional libraries: Intel MKL Sparse Blas and CuSparse. To be able to call the MKL sparse methods on the native side, we need to add its presets. The presets are generated by JavaCPP and define the signatures of each routine for which we need to link it through JNI (Java Native Interface) into a wrapper method in the Java side.

BLAS is organized into three sets of routines called *levels*:

Level 1 performs scalar, vector and vector-vector operations (dot product, norm,...).

Level 2 performs matrix-vector operations (matrix-vector multiplication (gemv)).

Level 3 performs matrix-matrix operations (general matrix multiplication (gemm)).

All the operations are accessible through a BLAS wrapper of *Nd4j*. They can be directly called via static methods or by calling them through an higher abstraction in *INDArray* objects.

This wrapper only wraps the methods of the dense BLAS library, this is why we need a specific wrapper for sparse BLAS library. However we want the different implementations to be transparent for the users, thus we use the dense wrapper as entry point and then redirect the call to the sparse wrapper internally. Thereby the user does not need to know the type of array is manipulating when he operates on it.

A wrapper contains the implementations of each level and Lapack, which is a linear equation system resolver library. The *Levels* interface define the set of operations for each level and then the implementations gather the arguments and link to the JavaCPP presets and the native side.

Similarly the sparse architecture has to have the same architecture: One sparse wrapper which contains the sparse implementations for each level.

Backends

Nd4j runs on top of interchangeable backends. The Nd4j API is build to be the same regardless of the current backend. This architecture allows us to swap the backend without changing the code: the written software can be easily deployed on multiple platforms with different architectures.

In-Place Routines

Some of the operations are performed in-place, that means the input matrix is modified during the operations.

We might have an issue when the number of non-zero values increases. We could run out of space in the different buffers because the memory can not be reallocated from the native side and because we can not know how many new values will be added. To avoid this problem we have to ensure having enough space before performing the operations.

The solution is to allocate the maximum size that the routine could use to the *DataBuffers*. At first it may look like inefficient and counterproductive according to the benefits we want to gain with the sparse implementation. However such operations are never executed on the whole array but only one a small sub-part of it.

Example

To illustrate this issue let's take an array $myArray$ with a size equals to $(100'000 \times 100'000 \times 100)$. The following operation takes a slice of the array $myArray$ (which corresponds in a row in this example) and assigns 1.0 to each value.

```
myArray.get(NDIndexArray.point(0),
    NDIndexArray.point(0),
    NDIndexArray.all())
    .assign(1.0),
```

Despite the huge size of the array, we only add at most 100 new values. If the buffers do not have the capacity to get this amount of new values, we need to reallocate them prior the operation.

Level 1 Routines

Sparse BLAS Level 1 routines in Intel MKL library [7] are a set of functions that perform vector operations on sparse vectors stored in a compressed format.

The vector is represented in the compressed form by two arrays, *values* and *indexes*. Figure 6.1 shows how a vector is represented in that format.

$$V_{(N\times 1)} = \begin{bmatrix} a_{k_0} \\ 0 \\ a_{k_1} \\ \dots \\ a_{k_{NNZ}} \end{bmatrix} \quad \rightarrow \quad \begin{matrix} Value = \begin{bmatrix} a_{k_0} & a_{k_1} & \dots & a_{k_{NNZ}} \end{bmatrix} \\ Indexes = \begin{bmatrix} k_0 & k_1 & \dots & k_{NNZ} \end{bmatrix} \end{matrix}$$

Figure 6.1: A vector v and its compressed form representation

The *values* array can be easily obtained from the CSR and COO format since it corresponds to the *values* buffer in both representations.

In case of CSR vector the *indexes* array corresponds to the column indexes buffer and can also be easily obtained.

However a value in a COO vector is always defined by two coordinates. It is due to a Nd4j singularity where the lowest possible rank for an array is 2, even in case of vector and scalar. The *indexes* array has to be extract from the *indexes* buffer depending whether it is a row vector or a column vector.

Once we have the two buffers, the procedure to call the BLAS routines is identical regardless of the underlying sparse format of the array. That mean we only need one implementation for all the sparse formats.

Level 2 and Level 3 Routines

The level 2 routines perform operations between a sparse matrix and dense vectors whereas level 3 perform between a sparse matrix and dense matrices. The operations between two sparse matrices are not supported by Intel MKL.

Intel MKL library supports several sparse matrix storage formats such as COO and CSR and each type has its own implementation of the routines.

For now in Nd4j only the general matrix-vector product is implemented (Gemv) for the COO format. All the input parameters of the COO routines can be extracted by a *SparseCOOGemv-Parameters* object which makes the future implementation of the level 2 routines easier.

Gemv Routine

This routine computes the matrix-vector product of a COO matrix and a dense vector. It can either compute $y = A * x$ or $y = A^T * x$ according to the parameter received in argument.

All the parameters needed to call the BLAS routine are extracted and computed according to the type of sparse format. In case of COO, we need an array of values, an array with the row indexes and an array with the column indexes. Since the indexes are flattened into one buffer, we split the indexes buffer to create a new array containing the row indexes and a second array containing the column indexes.

Libnd4j

In practice, Dl4j does not use all the BLAS routines. Most of the operation are reimplemented in Libnd4j which uses BLAS routines such as *dot*, *axpy*, *gemv* and *gemm* to optimized them. Those methods are picked up by JavaCPP during the compilation of Nd4j.

Then we have to provide an implementation of each methods. During the compilation of Libnd4j, all the implementations are grouped in a unique C++ class which are accessible through a big *switch* branch. Each operation has an unique identifier which is used to select the right operation implementation.

7 Results

Storing a Huge Array is now Possible

Having a sparse COO tensor representation makes possible to store a big array into the memory that would not fit in the dense representation.

For example, the following operation tries to create an array with a shape [10000, 10000, 100]. This array contains 10 billion of 0.

```
INDArray dense = Nd4j.zero(new int[]{10000, 10000, 100});
```

When executing this operation, Nd4j throws a *OutOfMemory* exception. The data is too big and required memory size can not be allocated.

```
java.lang.OutOfMemoryError: Cannot allocate new FloatPointer(10000000000):
  totalBytes = 1, physicalBytes = 208M
```

Ten billion of float numbers need 10 billion bit = 640 billion bit = 80 GB of RAM.

However if we store those 10 billion of 0 into the sparse COO format, the size is highly reduced. The values buffer and the indexes buffers are empty.

Assuming we have the same tensor but with a sparsity of 0.01, we only need:

1. 100 billion of values × sparsity = 100 million of values
   Storing 100 million of Float numbers requires: 64 million bit = 0.8 GB.

2. Each value has three coordinates, we need 100 million × 3 × 32bit = 1.2 GB.

At total, the sparse representation requires 2 GB of memory which is more acceptable compared to the average memory size of the current computers.

8 Future

Operations

All the level 1 operations are available but the level 2 and 3 still have to be implemented. The backbone of the implementation is done, we only need to extract the arguments from the array and call the sparse routines.

Regarding the native operations, everything still needs to be done. We need to implement each method into libnd4j and create the binding with Nd4j.

The operations on the tensors work using contractions. A contraction is a block or a slice along a dimension of the tensor with a rank lower than the rank of the original array. If we reduce the rank until we get a matrix, we can use the accelerated matrix operations. We perform the tensors operators by blocks.

Support of the GPU backend

Currently, only the backbone of the sparse GPU backend has been added in Nd4j. We need to generate and add the JavaCPP presets for CuSparse and then we will be able to link the API methods to the Cuda methods.

Make the Sparse Array Compliant with the API

Most of the methods of the INDArray interface are still to be implemented. We want the sparse representation to be compatible and interchangeable with the dense array. It's important that both representations have the same behavior and same features.

Support More Sparse Formats

An additional step would be to support different type of sparse format.

Due to the necessity of having a sparse representation that is not limited to the matrices and vectors, the CSR implementation has been interrupted in favor of the COO format. The CSR representation needs to be completed: Not all the indexes are supported, it misses the BLAS levels 2 and 3, native operations, GPU backend support.

Once the CSR format implemented, it will be easy to extend it to CSC since only the order is different between the two formats.

Then we will need to make the different formats compatible between them, with conversion methods from a format to another. This will allow getting two-dimensional slices of tensors in a different format to benefit from the advantages of each format.

Elgohary [8] developped an alternative version of CSR

Tensor Contraction Indexing

Currently, the mechanism to reverse an index and get the value (as explained in section 5.4.2) used a binary search. Sadly it does not provide an equal complexity in case of tensor contraction. A tensor contraction is a lower rank view of a tensor. Since the indexes buffer is sorted along the first dimension in a lexicographical order, it makes the task to access values of a contraction which does not contain the first dimension harder since the indexes are not sorted.

We could think to several possibility:

1. Extend the sorting mechanism to sort the indexes buffer according to a ordering of dimension. We could imagine a sort function to which we pass an array with the ordering of the dimensions such as:

   ```
   sortAlongDimension(indexesBuffer, new int[]{2, 0, 1});
   ```

   In this example the coordinates within the buffer would be inverted:
   $(i_0, i_1, i_2) \to (i_2, i_0, i_1)$ and then would be sorted along the new dimension ordering.

2. Use a more complex data-structure to store the indexes such as a skip list or a tree. A skip list is a multi-level linked list. We could take advantages of the levels to index each dimension and be able to have a more efficient random access to a value given a set of coordinates or to tensor contraction values.

The goal is to make the complexity constant regardless of the contraction dimensions.

Optimizations

Nd4j is built with the idea to avoid the JVM environment for storing the data. It is based on the postulate that data is usually huge and does not fit into the memory. However, with the new sparse implementation, we can store huge datasets into a reasonable size of memory (as long as it has a high sparsity).

We should study the possibility of storing sparse arrays on-heap with a data structure in managed memory such as ArrayList. A further idea would be to decompose the arrays and storing by block. The blocks with a high sparsity could be stored as on-heap in a sparse format, while the dense blocks would be stored off-heap as dense.

But the question about how on-heap data would interact with the Cuda remains open. Perhaps we could have a mixed backend which performs dense operations on GPU and sparse operations on CPU.

9 Conclusion

This work presents an implementation of a Compressed Row format (CSR) for sparse scalar, vector, and matrix. We also implemented the Coordinate format (COO) for sparse scalar, vector, matrix, and tensor.

To be able to perform accelerated sparse linear algebra operations we bound the Nd4j operation calls to the routines of Intel MKL Sparse BLAS for the two implemented formats.

We compared the memory needed to store a sparse data set using the dense representation and the COO representation. We observed significant improvements: the sparse format allows us storing some big data set that would not fit in memory if they used the dense representations.

A Appendices

Algorithm Execution Example

Sparse Offset Computation Algorithm

This appendix presents an execution of the algorithm 3 presented in section 5.4.6.

Let's start with an 4-order tensor $myTensor$ with a shape equal to $[2,4,4,5]$ on which we are calling the following operation:

```
myTensor.get(NDArrayIndex.all(),
NDArrayIndex.point(0),
NDArrayIndex.point(3),
NDArrayIndex.all());
```

Assuming that $myTensor$ is a view of a 5-order tensor and has a the following parameters:

```
sparseOffsets = [1, 1, 0, 0, 0]
flags = [1, 0, 0, 0, 0]
```

Assuming we name the dimension as [book, page, row, column], we are taking each column of the last row of the first pages of each book.

The indexes resolution returns the following parameters:

```
offsets = [0, 0]
shape =  [2, 5]
offset = 15
```

First step is to iterate over the dimension:
Iteration: i=0
Number of element in one book: $numElement = 4 \times 4 \times 5 = 60$
Book offset = $\lfloor offset/numElement \rfloor = \lfloor 15/60 \rfloor = 0$
Then we update the offset: $offset = offset - 0 * 60$
Iteration: i=1
Number of element in one page = $numElement = 4 \times 5 = 20$

Page offset = $\lfloor offset/numElement \rfloor = \lfloor 15/20 \rfloor = 0$
Then we update the offset: $offset = 15 - 0 * 20$
Iteration: i=2
Number of element in one row = $numElement = 5$
Page offset = $\lfloor offset/numElement \rfloor = \lfloor 15/5 \rfloor = 3$
Then we update the offset: $offset = 15 - 3 * 5 = 0$

Finally we reach the last dimension:
Column offset = $offset \mod numElement = 0 \mod 5 = 0$

We get an temporary offsets array equal to $newOffsets = [0, 0, 3, 0]$. Now we need to merge with the existing offsets of $myTensor$.

Its first dimension (shelf) is fixed, so we copy its sparse offset :
$finalOffest[0] = myTensor.sparseOffset[0] = 1$
Its second dimension (book) is active and there is already an non-zero offset. The offset is equal to
$finalOffset[1] = newOffset[0] + myTensor.sparseOffset[1] = 0 + 1 = 1$
Its third dimension (page)is active. The offset is equal to
$finalOffset[2] = newOffset[1] + myTensor.sparseOffset[2] = 0 + 0 = 0$
Its fourth dimension (row) is active. The offset is equal to
$finalOffset[3] = newOffset[2] + myTensor.sparseOffset[3] = 3 + 0 = 0$
Its fifth dimension (column) is active. The offset is equal to
$finalOffset[4] = newOffset[3] + myTensor.sparseOffset[4] = 0 + 0 = 0$

We finally get the final sparseOffset : $[1, 1, 0, 3, 0]$

Indexes Translation Algorithm

We present an execution of the algorithm 2 presented in section 5.4.5

Let's start with a 3-order tensor $myTensor$ which is a view with the following parameters:

```
shape = [1, 2, 2]
flags = [1, 0, 0]
hiddenDimension = [0]
sparseOffsets = [0, 1, 1]
```

The figure A.1 shows the tensor $myTensor$ in grey: it's a sup-part of a $2 \times 3 \times 3$ from which we performed the following operation:

```
myTensor = original.get(NDArrayIndex.newAxis(),
    NDArrayIndex.point(0),
    NDArrayIndex.interval(1, 3),
    NDArrayIndex.interval(1, 3));
```

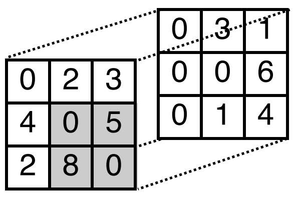


Figure A.1: A $2 \times 3 \times 3$ tensor with a view in grey

Let's translate the coordinates of the value 5. In $myTensor$ they are equal to $[0,0,1]$ while in $original$ tensors they are equal to $[0,1,2]$.

First we start the initilizing phase: the *result* array has a length of three (equal to the dimensionality of $original$). *idxPhy* is used to keep track of the position in the result array we are computing. *hidden* counts the number of hidden dimension we have encountered in the view.

Then we can start with the iterations: we iterate over all the dimensions of the view (idxVir keeps track of the position of the dimension).

Iteration: idxVir=0
The first condition is satisfied since $hidden = 0$ is smaller than the number of hidden dimension of $myTensor$ and its hidden dimension is actually the first 0. In this case this coordinate do not appear in the result, we can skip it and increment $hidden$ to 1

Teration: idxVir=1
The first condition is not satisfied: we have already encountered all the hidden dimensions of $myTensor$.

Then we start iterating as long as the dimension is fixed:

1. We have $idxPhy = 0$, according to the *flags* of the original array, $flags[0] = 1$ which means this dimension of the original array is fixed. We can add the $sparseOffset[0] = 0$ to the result.

   We have $result = [0,\emptyset,\emptyset]$ and $idxPhy = 1$

2. The second dimension is not fixed ($flags[idxPhy] = 0$). We stop iterating.

The second dimension is not fixed ($flags[idxPhy] = 0$). We add $sparseOffsets[idxPhx] + viewIdx[idxVir] = 1 + 0 = 1$ into the result.

We have $result = [0,1,\emptyset]$, $idxPhy = 2$ and $idxVir = 2$

Iteration: idxVir=2
This dimension is neither hidden nor fixed. We add $sparseOffsets[idxPhx] + viewIdx[idxVir] = 1 + 1 = 2$ into the result.

We finally get the output: $result = [0,1,2]$

Code Snippets

Extract a sub-array of a CSR matrix

Listing A.1: Extract a sub-array of a CSR matrix

```java
    public INDArray subArray(ShapeOffsetResolution resolution) {

        long[] offsets = resolution.getOffsets();
        int[] shape = LongUtils.toInts(resolution.getShapes());


        List<Integer> accuColumns = new ArrayList<>();
        List<Integer> accuPointerB = new ArrayList<>();
        List<Integer> accuPointerE = new ArrayList<>();

        if (shape.length == 2) {

            if (resolution.getOffset() != 0) {
                offsets[0] = (int) resolution.getOffset() / shape()[1];
                offsets[1] = (int) resolution.getOffset() % shape()[1];
            }
            long firstRow = offsets[0];
            long lastRow = firstRow + shape[0];
            long firstElement = offsets[1];
            long lastElement = firstElement + shape[1];

            int count = 0;
            int i = 0;
            for (int rowIdx = 0; rowIdx < lastRow; rowIdx++) {

                boolean isFirstInRow = true;
                for (int idx = pointerB.getInt(rowIdx); idx < pointerE.getInt(
                  rowIdx); idx++) {

                    int colIdx = columnsPointers.getInt(count);

                    // add the element in the subarray it it belongs to the view
                    if (colIdx >= firstElement && colIdx < lastElement && rowIdx
                      >= firstRow && rowIdx < lastRow) {

                        // add the new column pointer for this element
                        accuColumns.add((int) (colIdx - firstElement));

                        if (isFirstInRow) {
                            // Add the index of the first element of the row in
                              the pointer array
                            accuPointerB.add(idx);
                            accuPointerE.add(idx + 1);
                            isFirstInRow = false;
                        } else {
                            // update the last element pointer array
                            accuPointerE.set((int) (rowIdx - firstRow), idx + 1);
                        }
                    }
                    count++;
                }

                // If the row doesn't contain any element and is included in the
```

```
                    selected rows
                if (isFirstInRow && rowIdx >= firstRow && rowIdx < lastRow) {
                    int lastIdx = i == 0 ? 0 : accuPointerE.get(i - 1);
                    accuPointerB.add(lastIdx);
                    accuPointerE.add(lastIdx);
                }
                if (rowIdx >= firstRow && rowIdx <= lastRow) {
                    i++;
                }
            }

            int[] newColumns = Ints.toArray(accuColumns);
            int[] newPointerB = Ints.toArray(accuPointerB);
            int[] newPointerE = Ints.toArray(accuPointerE);

            INDArray subarray = Nd4j.createSparseCSR(values, newColumns,
              newPointerB, newPointerE, shape);

            return subarray;

        } else {
            throw new UnsupportedOperationException();
        }
    }
```

Put a Value into a CSR matrix

Listing A.2: Put a new value into a CSR array

```
        public INDArray putScalar(int row, int col, double value) {

            int idx = pointerB.getInt(row);
            int idxNextRow = pointerE.getInt(row);

            while (columnsPointers.getInt(idx) < col && columnsPointers.getInt(
              idx) < idxNextRow) {
                idx++;
            }
            if (columnsPointers.getInt(idx) == col) {
                values.put(idx, value);
            } else {
                //Add a new entry in both buffers at a given position
                values = addAtPosition(values, length, idx, value);
                columnsPointers = addAtPosition(columnsPointers, length, idx, col
                  );
                length++;

                // shift the indices of the next rows
                pointerE.put(row, pointerE.getInt(row) + 1);
                for (int i = row + 1; i < rows; i++) {
                    pointerB.put(i, pointerB.getInt(i) + 1);
                    pointerE.put(i, pointerE.getInt(i) + 1);
                }
            }
            return this;
        }
```

Add a value into a buffer

Listing A.3: Add a value into a buffer

```
private DataBuffer addAtPosition(DataBuffer buf, long dataSize, int pos,
  double value) {

    DataBuffer buffer = (buf.length() == dataSize) ? reallocate(buf) :
      buf;
    double[] tail = buffer.getDoublesAt(pos, (int) dataSize - pos);
    buffer.put(pos, value);

    // we have to shift right the tail
    for (int i = 0; i < tail.length; i++) {
        buffer.put(i + pos + 1, tail[i]);
    }
    return buffer;
}
```

Acknowledgements

I would like to thank Professor Jäggi and François Garillot for supervising me during my work and giving the opportunity to accomplish my Master thesis at Skymind. I would also like to thank the Skymind team for the help providing and particularly Raver for being always available to answer my questions and for helping me get a clear and deep understanding of Nd4j. Finally, I would like to thank the SF-based Skymind team for having welcomed me for the 6 months.

San Francisco, August 17, 2017

Bibliography

[1] “Zipf’s law.” https://en.wikipedia.org/wiki/Zipf

[2] “Netflix grand prize.” http://www.netflixprize.com. [Online; accessed 8-August-2017].

[3] Y. Koren, “1 the bellkor solution to the netflix grand prize,” 2009.

[4] “Intel MKL sparse format.” https://software.intel.com/en-us/mkl-developer-reference-c-sparse-blas-csr-matrix-storage-format. [Online; accessed 8-August-2017].

[5] “Breeze - a numerical processing library for scala.” https://github.com/scalanlp/breeze. [Online; accessed 14-August-2017].

[6] “Numpy.” http://www.numpy.org/. [Online; accessed 15-August-2017].

[7] “Intel MKL sparse level 1 routines.” https://software.intel.com/en-us/mkl-developer-reference-c-sparse-blas-level-1-routines. [Online; accessed 13-August-2017].

[8] A. Elgohary, M. Boehm, P. J. Haas, F. R. Reiss, and B. Reinwald, “Compressed linear algebra for large-scale machine learning,” *Proc. VLDB Endow.*, vol. 9, pp. 960–971, Aug. 2016.