Low Level Big Data Processing

Jaime Salvador-Meneses[1], Zoila Ruiz-Chavez[1] and Jose Garcia-Rodriguez[2]
[1]*Universidad Central del Ecuador, Ciudadela Universitaria, Quito, Ecuador*
[2]*Universidad de Alicante, Ap. 99. 03080, Alicante, Spain*

Keywords: Big Data, Compression, Processing, Categorical Data, BLAS.

Abstract: The machine learning algorithms, prior to their application, require that the information be stored in memory. Reducing the amount of memory used for data representation clearly reduces the number of operations required to process it. Many of the current libraries represent the information in the traditional way, which forces you to iterate the whole set of data to obtain the desired result. In this paper we propose a technique to process categorical information previously encoded using the bit-level schema, the method proposes a block processing which reduces the number of iterations on the original data and, at the same time, maintains a processing performance similar to the processing of the original data. The method requires the information to be stored in memory, which allows you to optimize the volume of memory consumed for representation as well as the operations required to process it. The results of the experiments carried out show a slightly lower time processing than the obtained with traditional implementations, which allows us to obtain a good performance.

1 INTRODUCTION

The number of attributes (also called dimension) in many datasets is large, and many algorithms do not work well with datasets that have a high dimension. Currently, it is a challenge to process data sets with a high dimensionality such as censuses conducted in different countries (Rai and Singh, 2010).

Latin America, in the last 20 years, has tended to take greater advantage of census information (Feres, 2010), this information is mostly categorical information (variables that take a reduced set of values). The representation of this information in digital media can be optimized given the categorical nature.

A census consists of a set of *m* observations (also called records) each of which contains *n* attributes. An observation contains the answers to a questionnaire given by all members of a household (Bruni, 2004).

This paper proposes a mechanism for processing categorical information using bit-level operations. Prior to processing, it is necessary to encode (compress) the information into a specific format. The mechanism proposes compressing the information into packets of a certain number of bits (16, 32, 64 bits), in each packet a certain number of values are stored.

Bitwise operations (AND, OR, etc.) are an important part of modern programming languages because they allow you to replace arithmetic operations with more efficient operations (Seshadri et al., 2015).

This document is organized as follows: Section 2 summarizes the standard that defines the algebraic operations that can be performed on data sets as well as some of the main libraries that implement it, Section 3 presents an alternative for information processing based on the BLAS Level 1 standard, Section 4 presents several results obtained using the proposed processing method and, finally, Section 5 presents some conclusions.

2 BLAS SPECIFICATION

In this section we present a summary of some libraries that implements the BLAS specification and a brief review about encoding categorical data.

Basic Linear Algebra Subprograms (BLAS) is a specification that defines low-level routines for performing operations related to linear algebra. Operations are related to scalar, vector and matrix process. BLAS define 3 levels:

- *BLAS Level 1* defines operations between vector scales
- *BLAS Level 2* defines operations between scales

Salvador-Meneses, J., Ruiz-Chavez, Z. and Garcia-Rodriguez, J.
Low Level Big Data Processing.
DOI: 10.5220/0007227103470352
In *Proceedings of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2018) - Volume 1: KDIR*, pages 347-352
ISBN: 978-989-758-330-8

and matrices

- *BLAS Level 3* defines operations between scales, vectors and matrices

This document focuses on the BLAS Level 1 specification.

2.1 BLAS Level 1

This level specifies operations between scales and vectors and operations between vectors. Table 1 lists all the functions described in BLAS Level 1.

2.2 Libraries that Implement BLAS

This section describes some of the libraries that implement the BLAS Level 1 specification. The approach is based on implementations without GPU graphics acceleration.

All the libraries described below make extensive use of vector and matrix operations and therefore require an optimal representation of this type of algebraic structure (vectors and matrices).

Most of the libraries described in this section represent the information as arrangements of *float* or *double* elements, so the size in bytes needed to represent a vector is:

$$total = number\,of\,elements * 4$$

2.2.1 ViennaCL

ViennaCL's approach is to provide a high-level abstraction using C++ for the data represented on a GPU (Rupp et al., 2010). To work with GPUs, ViennaCL maintains two approaches, the first based on CUDA [1], while the second is based on OpenCL [2].

In the event that the underlying system running ViennaCL does not support GPU acceleration, ViennaCL makes use of the multi-processing associated with the processor (CPU).

ViennaCL is extensible, the data types provided can be invoked from other libraries such as Armadillo and uBLAS (Rupp et al., 2016). The information is represented in a scheme similar to STL [3], for the case of vectors the data type is:

$$vector < T, aligment >$$

where T can be a primitive type like *char*, *short*, *int*, *long*, *float*, *double*.

[1] https://developer.nvidia.com/cuda-zone

[2] http://www.khronos.org/opencl/

[3] https://isocpp.org/std/the-standard

2.2.2 uBLAS

uBLAS is a C++ library which provides support for dense, packed and sparse matrices [4]. uBLAS is part of the BOOST project and corresponds to a CPU implementation of the BLAS standard (all levels) (Tillet et al., 2012).

The information is represented in a scheme similar to STL, for the case of vectors the data type is:

$$vector < T >$$

where T can be a primitive type like *char*, *short*, *int*, *long*, *float*, *double*.

2.2.3 Armadillo

Armadillo is an open source linear algebra library written in C++. The library is useful in the development of algorithms written in C++. Armadillo provides efficient, object-like implementations for vectors, arrays, and cubes (tensors) (Sanderson and Curtin, 2016).

There are libraries that use Armadillo for its implementation, such as `MLPACK`[5] which is written using the matrix support of Armadillo (Curtin et al., 2012).

Armadillo represents arrays of elements using a column or row oriented format using the classes *Col* and *Row* respectively. The two types of data need as a parameter the type of data to be represented within the vector, these types of data can be: uchar, unsigned int (u32), int (s32), unsigned long long (u64), long long (s64), float, double[6].

The size in bytes needed to represent a vector corresponds to:

$$total = number\,of\,elements * sizeof(element\_type)$$

2.2.4 LAPACK

Linear Algebra Package (LAPACK) is a library of routines written in Fortran which implements the functionality described in the BLAS specification. There are additional modules which allow the library to be used from C++ or other programming languages.

LAPACK is the industry standard for interacting with linear algebra software (Cao et al., 2014), it supports the BLAS Level 1 functions using native vectors of the programming language used, C++ in this

[4] http://www.boost.org/doc/libs/1_65_1/libs/numeric/ublas/doc/index.html

[5] http://mlpack.org/index.html

[6] u64 and s64 supported only in 64 bit systems

Table 1: BLAS Level 1 functions.

Function	Description	Mathematical Formula
Swap	Exchange of two vectors	$y \leftrightarrow y$
Stretch	Vector scaling	$x \leftarrow \alpha x$
Assiggment	Copy a vector	$y \leftarrow x$
Multiply add	Sum of two vectors	$y \leftarrow \alpha x + y$
Multiply subtract	Subtraction of two vectors	$y \leftarrow \alpha x - y$
Inner dot product	Dot product between two vectors	$\alpha \leftarrow x^T y$
L^1 norm	L^1 norm	$\alpha \leftarrow \|\|x\|\|_1$
L^2 norm	Vector 2-norm (Euclidean norm in $\mathcal{R}^2$)	$\alpha \leftarrow \|\|x\|\|_2$
L^∞ norm	Infinite norm	$\alpha \leftarrow \|\|x\|\|_\infty$
L^∞ norm index	Infinite-index norm	$i \leftarrow max_i \|x_i\|$
Plane rotation	Rotation	$(x,y) \leftarrow (\alpha x + \beta y, -\beta x + \alpha y)$

Table 2: Armadillo suported vector format.

Input vector type	Armadilo type	Col	Row
unsigned char	-	uchar_vec	uchar_rowvec
unsigned int	u32	u32_vec	u32_rowvec
int	s32	s32_vec	s32_rowvec
unsigned long long	u64	u64_vec	u64_rowvec
long long	s64	s64_vec	s64_rowvec
float	-	fvec	frowvec
double	-	vec,dvec	rowvec,drowvec

case. Additionally, it supports operations with complex numbers.

Table 3 shows the detail of the *norm2* function and the prefix used to name the functions (*cblas_Xnrm2* where *X* represents the prefix)[7].

Table 3: LAPACK supported vector format.

Input vector	Result type	Prefix
Vector of float	float	S
Vector of double	double	D
Vector of complex float	float	SC
Vector of domplex double	double	DZ

As shown in the above table, vectors are represented as arrays of *float* or *double* elements, so the size in bytes needed to represent a vector is:

$$total = number\,of\,elements * 4$$

2.3 Current Compression Algorithms

As shown in the previous sections, the libraries described above work with uncompressed data. In most cases the information is represented as one-dimensional vectors containing 32-bit elements (float, int). When working with categorical data, there are some options for working with this type of information.

[7]https://www.ibm.com/support/knowledgecenter/en/SSFHY8_5.5.0/com.ibm.cluster.essl.v5r5.essl100.doc/am5gr_hsnrm2.htm

Some options for compressing/representing one-dimensional (vector) data sets are described below.

Run-length Encoding. Consecutive data streams are encoded using a $(key, value)$ pair (Elgohary et al., 2017).

Offset-list encoding. Unlike the previous method, correlated pairs of data are encoded (Elgohary et al., 2017).

GZIP. This type of method overloads the CPU when decompressing the data (Chen et al., 2001).

Bit-level Compression. A data set is packaged in 32-bit blocks (De Grande, 2016).

3 PROCESSING APPROACH

In this section we propose a new mechanism for processing categorical data, the processing method proposed corresponds to a variation of the processing of data compressed using **bit level compression** method described in Section 2.3. The processing method developed corresponds to a process of bit-level elements with bit-shifting operations.

3.1 Algorithms

The algorithm consists of representing within 4-bytes n-categorical values of a variable. This means that to access the value of a particular observation, a double indexing is necessary:

1. Access the index of the value (4-bytes) containing the searched element.
2. Index within 32 bits to access the value.

For the implementation of the proposed method we use a mixture of arithmetic and bit-wise operations, specifically:

- Logical AND (&)
- Logical OR (|)
- Logical Shift Left ($\ll$)
- Logical Shift Right ($\gg$)

As an example, in this section we implement the L^2 norm algorithm. As you can see, all the functions described in the BLAS Level 1 can be implemented in the same way.

3.1.1 L^2 Norm Algorithm

Algorithm 1 represents the algorithm for calculating the L^2 norm of a compressed vector. It should be noted that for each iteration on the compressed vector (*vector*), n-elements represented in 32 bits of data are accessed.

The input values correspond to:

- vector: Input vector stored as compressed vector of *dataSize* bits.
- size: Vector size.
- dataSize: Bit size used for representation.

The algorithm iterates over each element of the compressed vector and then iterates inside each block. The number of elements contained in each block corresponds to:

$$elementsPerBlock \leftarrow 32/dataSize$$

It is possible to optimize the iteration within the block by performing bit-wise operations. The Algorithm 2 shows the implementation of the same algorithm using bit-wise operations. For a given *dataSize*, a code block is generated that operates on the bits in such a way that iteration on the block is avoided.

The Algorithm 2 ahows the Algorithm 1 modified.

Algorithm 1: Calculate L^2 norm - version 1.

Data: vector, size, dataSize

$elementsPerBlock \leftarrow 32/dataSize$;
$mask \leftarrow$ sequence of dataSize-bits with value = 1
$sum \leftarrow 0$;
for $index \leftarrow 0$ **to** $size-1$ **do**
  $value \leftarrow vector1[index]$;
  for $i \leftarrow 0$ **to** $elementsPerBlock-1$ **do**
    $v \leftarrow value1 \gg (i * dataSize)$ & $mask$;
    $sum \leftarrow sum + v * v$;
  end
end
$norm2 \leftarrow \sqrt{sum}$

4 EXPERIMENTS

This section presents the result of the processing of random generated vectors. The memory consumption of the representation of the data in the main memory of a computer was not analyzed, instead the processing speed was measured.

All libraries were tested on Ubuntu 16.04 LTE using GCC 5.0.4 as compiler with the default settings. None were compiled with optimizations for the test platform.

4.1 Test Platform

Several vector sizes were considered for testing: 10^3, 10^4,..., 10^9 elements. The test data set contains randomly generated items in the range $[0.119]$. All the libraries mentioned in the Section 2.2 use vector representation with float-type elements.

The platform on which the tests were performed corresponds to:

- Procesador: Intel(R) Core(TM) i5-5200 CPU
- Processor speed: 2.20GHz
- RAM Memory: 16.0GB
- Operating System: Ubuntu 16.04LTE 64 bits
- Compiler: GCC 5.04 64bits

Some libraries provide particular implementations of the vector data type:

- LAPACK uses native `float *` implementation
- uBLAS uses the `ublas::vector<float>` implementation
- Armadillo uses the `arma::vec` implementation

Algorithm 2: Calculate L^2 norm - version 2.

```
Data: vector, size, dataSize
function SUM_06_SQUARE(int b, int mask )
  → int
{
    b1 ← ((b >> 0) & mask);
    b2 ← ((b >> 6) & mask);
    b3 ← ((b >> 12) & mask);
    b4 ← ((b >> 18) & mask);
    b5 ← ((b >> 24) & mask);
    return b1*b1 + b2*b2 + b3*b3 + b4*b4
  + b5*b5;
}
function SUM_07_SQUARE(int b, int mask )
  → int
{
    b1 ← ((b >> 0) & mask);
    b2 ← ((b >> 7) & mask);
    b3 ← ((b >> 14) & mask);
    b4 ← ((b >> 21) & mask);
    return b1*b1 + b2*b2 + b3*b3 + b4*b4;
}
elementsPerBlock ← 32/dataSize;
mask ← sequence of dataSize-bits with value
  = 1
sum ← 0;
nn ← dataSize;
for index ← 0 to size − 1 do
    value ← vector1[index];
    sum ←
      sum + SUM_nn_SQUARE(value, mask);
end
norm2 ← √sum
```

- ViennaCL uses the STL `std::vector<float>` implementation

The function that were verified correspond to L^2 norm of the BLAS Level 1 (operations between vectors and scalars). Below are some results obtained, from the second graph the values of ViennaCL for vectors of size 10^9 were omitted due to the considerable time needed for the process.

4.2 Results: L^2 Norm

This section presents results obtained when performing operations with vectors represented in traditional formats (Section 2.2) and in compressed format. For testing purposes, a vector similar to the one described in the previous section is used.

Figure 1 and Table 4 show the time taken to calculate the L^2 norm for vectors of different sizes.

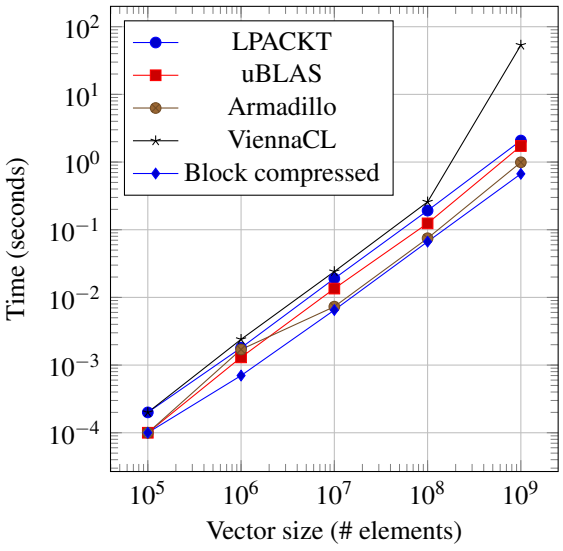

Figure 1: L^2 - Processing time.

As we can see, the block compressed approach has a better performance than the others methods and is similar to *Armadillo* in performance but it uses low memory. We can expect the other algebraic functions to behave similarly.

5 CONCLUSIONS

In this document we reviewed some of the most common libraries which implement the algebraic operations that constitute the core for the implementation of machine learning algorithms. In general, it can be concluded that the categorical information processing proposed slightly reduces the processing time of the information compared to similar implementations under the BLAS Level 1 standard. The encoded data representation allows to reduce the amount of memory needed to represent the information as well as the number of iterations over the data.

In the case of the test dataset, the calculation of the L^2 norm showed a slight reduction in the processing time (for more details, see Table 4). In the case of operations involving two vectors, it is recommended to encode both vectors using the same scheme.

Future work is proposed to (1) implement all the functions described in the BLAS Level 1 standard and test more sophisticated functions like matrix operations, (2) extend representation and processing to matrix data structures and (3) implement the representation and processing using parallel programming with Graphics Processing Unit (GPU).

Table 4: L^2 norm - Processing time.

	Time (seconds)				
Vector size (# elements)	LAPACK	uBLAS	Armadillo	ViennaCL	Bit by bit
10^3	0	0	0	0	0
10^4	0	0	0	0	0
10^5	0.0002	0.0001	0.0001	0.0002	0.0001
10^6	0.0018	0.0013	0.0017	0.0024	0.0007
10^7	0.0190	0.0135	0.0073	0.0241	0.0065
10^8	0.1918	0.1242	0.0748	0.2576	0.0666
10^9	2.0787	1.7273	0.9916	53.4602	0.6696

ACKNOWLEDGEMENTS

The authors would like to thank to Universidad Central del Ecuador and its initiative called *Programa de Doctorado en Informática* for the support during the writing of this paper. This work has been supported with Universidad Central del Ecuador funds.

REFERENCES

Bruni, R. (2004). Discrete models for data imputation. *Discrete Applied Mathematics*, 144(1-2):59–69.

Cao, C., Dongarra, J., Du, P., Gates, M., Luszczek, P., and Tomov, S. (2014). clMAGMA: High Performance Dense Linear Algebra with OpenCL. *Proceedings of the International Workshop on OpenCL 2013 {&} 2014*, pages 1:1—-1:9.

Chen, Z., Gehrke, J., and Korn, F. (2001). Query optimization in compressed database systems. *ACM SIGMOD Record*, 30(2):271–282.

Curtin, R. R., Cline, J. R., Slagle, N. P., March, W. B., Ram, P., Mehta, N. A., and Gray, A. G. (2012). MLPACK: A Scalable C++ Machine Learning Library. pages 1–5.

De Grande, P. (2016). El formato Redatam. *ESTUDIOS DEMOGRÁFICOS Y URBANOS*, 31:811–832.

Elgohary, A., Boehm, M., Haas, P. J., Reiss, F. R., and Reinwald, B. (2017). Scaling Machine Learning via Compressed Linear Algebra. *ACM SIGMOD Record*, 46(1):42–49.

Feres, J. C. (2010). XII . Medición de la pobreza a través de los censos de población y vivienda. pages 327–335.

Rai, P. and Singh, S. (2010). A Survey of Clustering Techniques. *International Journal of Computer Applications*, 7(12):1–5.

Rupp, K., Rudolf, F., and Weinbub, J. (2010). ViennaCL-a high level linear algebra library for GPUs and multi-core CPUs. *Intl.˜Workshop on GPUs and Scientific Applications*, pages 51–56.

Rupp, K., Tillet, P., Rudolf, F., and Weinbub, J. (2016). ViennaCL—Linear Algebra Library for Multi-and Many-Core Architectures. *SIAM Journal on*.

Sanderson, C. and Curtin, R. (2016). Armadillo: a template-based C++ library for linear algebra. *The Journal of Open Source Software*, 1:26.

Seshadri, V., Hsieh, K., Boroumand, A., Lee, D., Kozuch, M. A., Mutlu, O., Gibbons, P. B., and Mowry, T. C. (2015). Fast Bulk Bitwise and and or in DRAM. *IEEE Computer Architecture Letters*, 14(2):127–131.

Tillet, P., Rupp, K., and Selberherr, S. (2012). An Automatic OpenCL Compute Kernel Generator for Basic Linear Algebra Operations. *Simulation Series*, 44(6).