Ruby Association Grant 2020 Final Term Report

Author: Udit Gulati

Email: uditgulati0@gmail.com

Github: @uditgulati

Contents

- Overview
- Code
- Installation
- Results
- Pull requests
- Work Completed
- Future work

Overview

NumRuby is the re-implementation of popular dense matrix library NMatrix. It is re-implemented with primary focus being performance and scalability. It has shown upto 100x speedup for the fundamental functionalities such as elementwise operations. This project aims at adding Views support to the NumRuby library. Views are used to have multi-dimensional data structures exchange data without creating copies which saves a lot of memory and execution time.

Sparse matrices are an important part of scientific computing. Many popular modern languages like Python, Julia and R (some even have this as part of their standard libraries) have sparse matrices support while Ruby doesn't have a good library for sparse. Ruby-Sparse library is aimed to build an efficient and feature-rich end-user intended Sparse matrix library in Ruby with interfaces to popular dense matrix libraries (like Numo-Narray, NumRuby[NMatrix]) with linear algebra support.

Code

https://github.com/sciruby/ruby-sparse

https://github.com/sciruby/numruby

Installation

Ruby-Sparse

Build library

```
git clone https://github.com/sciruby/ruby-sparse
cd ruby-sparse/
gem install bundler
bundle install
rake compile
```

Try code

```
rake pry
```

NumRuby

Dependencies installation (Debian/Ubuntu)

```
sudo apt-get install libopenblas-dev
sudo apt-get install liblapack-dev
sudo apt-get install liblapacke-dev
```

Dependencies installation (Mac OSX)

```
brew install openblas
brew install lapack
```

Build library

```
git clone https://github.com/sciruby/numruby
cd numruby/
gem install bundler
bundle install
rake compile
```

Try code

```
rake pry
```

Code and Usage Overview

Ruby-Sparse work

COO implementation overview

```
typedef struct COO_STRUCT
{
  sp_dtype dtype;
  size_t ndims;
  size_t count;     //count of non-zero elements
  size_t* shape;
  double* elements; //elements array
  size_t* ia;       //row index
  size_t* ja;       //col index
}coo_matrix;
```

```
[1] pry(main)> n = RubySparse::COO.new [3, 3], [1, 2, 3], [0, 1, 2], [0, 1, 2]
=> "#<RubySparse::COO:0x0000562297c65f88; shape: [3,3]; dtype: sp_float64; nnz: 3>"
[2] pry(main)> n.elements
=> [1.0, 2.0, 3.0]
[3] pry(main)> n.coords
=> [[0, 1, 2], [0, 1, 2]]
[4] pry(main)> n.shape
=> [3, 3]
```

CSR implementation overview

```
typedef struct CSR_STRUCT
{
  sp_dtype dtype;
  size_t ndims;
  size_t count;     //count of non-zero elements
  size_t* shape;
  double* elements; //elements array
  size_t* ip;       //row pointer vals
  size_t* ja;       //col index
}csr_matrix;
```

CSR sparse representation compresses the row index array which leads to reduction in amount of data stored for large enough matrices. In this implementation, `elements` array represents the data values of the sparse matrix which is all the non-zero values in the sparse matrix. `ip` represents the compressed row array values which is the cumulative sum of number of elements in each row till the given row. `ja` represents the column index values.

```
[1] pry(main)> M = RubySparse::CSR.new [3, 3], [3, 2, 5], [0, 1, 2], [0, 1, 2]
=> "#<RubySparse::CSR:0x000055abb9f26718; shape: [3,3]; dtype: sp_float64; nnz: 3>"
[2] pry(main)> M.elements
=> [3.0, 2.0, 5.0]
[3] pry(main)> M.indptr
=> [0, 1, 2, 3]
[4] pry(main)> M.indices
=> [0, 1, 2]
[5] pry(main)> M.shape
=> [3, 3]
[6] pry(main)> M.dtype
=> :sp_float64
```

CSC implementation overview

```
typedef struct CSC_STRUCT
{
  sp_dtype dtype;
  size_t ndims;
  size_t count;     //count of non-zero elements
  size_t* shape;
  double* elements; //elements array
  size_t* ia;       //row index
  size_t* jp;       //col pointer vals
}csc_matrix;
```

CSC sparse representation compresses the column index array which leads to reduction in amount of data stored for large enough matrices. In this implementation, `elements` array represents the data values of the sparse matrix which is all the non-zero values in the sparse matrix. `jp` represents the compressed column array values which is the cumulative sum of number of elements in each column till the given column. `ia` represents the row index values.

```
[1] pry(main)> M = RubySparse::CSC.new [3, 3], [3, 2, 5], [0, 1, 2], [0, 1, 2]
=> "#<RubySparse::CSC:0x000055b9d6dadf78; shape: [3,3]; dtype: sp_float64; nnz: 3>"
[2] pry(main)> M.elements
=> [3.0, 2.0, 5.0]
[3] pry(main)> M.indptr
=> [0, 1, 2, 3]
[4] pry(main)> M.indices
=> [0, 1, 2]
[5] pry(main)> M.dtype
=> :sp_float64
```

DIA implementation overview

```
typedef struct DIA_STRUCT
{
  sp_dtype dtype;
  size_t ndims;
  size_t count;     //count of diagonal elements
  size_t* shape;
  double* elements; //elements array
}dia_matrix;
```

DIA sparse representation is used only for those sparse matrices that have non-zero elements only along the diagonal of the matrix. `elements` stores the non-zero values of the matrix along the diagonal. There is no need for index values if all the diagonal values are provided and in the order in which they appear in increasing order of row/column index.

```
[1] pry(main)> n = RubySparse::DIA.new [3, 3], [1, 0, 3]
=> #<RubySparse::DIA:0x2b06a3ecf69c>
[2] pry(main)> n.elements
=> [1.0, 0.0, 3.0]
[3] pry(main)> n.shape
=> [3, 3]
[4] pry(main)> n.dim
=> 2
[5] pry(main)> n.dtype
=> :sp_float64
```

NMatrix to/from Sparse conversion

```
def self.from_nmatrix(nmat)
  nm_elements = nmat.elements
  nm_shape = nmat.shape

  if nmat.dim != 2
    raise StandardError.new "NMatrix must be of 2 dimensions."
  end

  csr_elements = []
  csr_ia = []
  csr_ja = []

  for i in (0...nmat.shape[0])
    for j in (0...nmat.shape[1])
      nm_index = (nmat.shape[1] * i) + j
      nm_value = nm_elements[nm_index]
      if nm_value == 0
        next
      end
      csr_elements.append(nm_value)
      csr_ia.append(i)
      csr_ja.append(j)
    end
  end

  csr_mat = self.new nm_shape, csr_elements, csr_ia, csr_ja
  return csr_mat
end
```

```
def to_nmatrix
  ip = self.indptr
  ja = self.indices

  nm_elements = Array.new(self.shape[0]*self.shape[1], 0)
  for i in (0...self.shape[0])
    for j in (ip[i]...ip[i + 1])
      nm_index = (self.shape[1] * i) + ja[j]
      nm_elements[nm_index] = self.elements[j]
    end
  end

  m = NMatrix.new self.shape, nm_elements
  return m
end
```

Above are the NMatrix/CSR sparse matrix conversion methods. Similarly, the conversion with NMatrix for other sparse implementations (COO, CSC, DIA) have also been implemented and can be found in the pull requests provided below.

Elementwise operators

```
[1] pry(main)> n = RubySparse::COO.new [3, 3], [1, 2, 3], [0, 1, 2], [0, 1, 2]
=> "#<RubySparse::COO:0x0000559adcab2878; shape: [3,3]; dtype: sp_float64; nnz: 3>"
[2] pry(main)> m = RubySparse::COO.new [3, 3], [3, 2, 3], [0, 1, 2], [2, 2, 2]
=> "#<RubySparse::COO:0x0000559adcaed608; shape: [3,3]; dtype: sp_float64; nnz: 3>"
[3] pry(main)> x = n + m
=> "#<RubySparse::COO:0x0000559adc69bcf8; shape: [3,3]; dtype: sp_float64; nnz: 5>"
[4] pry(main)> x.class
=> RubySparse::COO
[5] pry(main)> x.elements
=> [1.0, 3.0, 2.0, 2.0, 6.0]
[6] pry(main)> x.coords
=> [[0, 0, 1, 1, 2], [0, 2, 1, 2, 2]]
[7] pry(main)> y = n.sin
=> "#<RubySparse::COO:0x0000559adc6d7a78; shape: [3,3]; dtype: sp_float64; nnz: 3>"
[8] pry(main)> y.class
=> RubySparse::COO
[9] pry(main)> y.elements
=> [0.8414709848078965, 0.9092974268256817, 0.1411200080598672]
[10] pry(main)> y.coords
=> [[0, 1, 2], [0, 1, 2]]
```

Iteration

```
[1] pry(main)> m = RubySparse::COO.new [3, 3], [3, 2, 3], [0, 1, 2], [2, 2, 2]
=> "#<RubySparse::COO:0x00005642595d9278; shape: [3,3]; dtype: sp_float64; nnz: 3>"
[2] pry(main)> m.each { |e| p e }
3.0
2.0
3.0
=> "#<RubySparse::COO:0x00005642595d9278; shape: [3,3]; dtype: sp_float64; nnz: 3>"
[3] pry(main)> m.each_with_indices { |e| p e }
[3.0, 0, 2]
[2.0, 1, 2]
[3.0, 2, 2]
=> "#<RubySparse::COO:0x00005642595d9278; shape: [3,3]; dtype: sp_float64; nnz: 3>"
[4] pry(main)> M = RubySparse::CSR.new [3, 3], [3, 2, 5], [0, 1, 2], [0, 1, 2]
=> "#<RubySparse::CSR:0x00005642596d3890; shape: [3,3]; dtype: sp_float64; nnz: 3>"
[5] pry(main)> M.each_row { |e| p e }
[3.0]
[2.0]
[5.0]
=> "#<RubySparse::CSR:0x00005642596d3890; shape: [3,3]; dtype: sp_float64; nnz: 3>"
[6] pry(main)> N = RubySparse::CSC.new [3, 3], [3, 2, 5, 4], [0, 1, 2, 1], [0, 0, 0, 2]
=> "#<RubySparse::CSC:0x000056425913a6e0; shape: [3,3]; dtype: sp_float64; nnz: 4>"
[7] pry(main)> N.each_column { |e| p e }
[3.0, 2.0, 5.0]
[]
[4.0]
=> "#<RubySparse::CSC:0x000056425913a6e0; shape: [3,3]; dtype: sp_float64; nnz: 4>"
```

NumRuby Views

```
typedef struct NMATRIX_STRUCT
{
  nm_dtype dtype;
  size_t ndims;
  size_t count;
  size_t* shape;
  void* elements;
}nmatrix;
```

```
typedef struct NMATRIX_BUFFER_STRUCT
{
  nm_dtype dtype;
  size_t count;
  size_t ndims;
  size_t* shape;
  void* buffer_ele_start_ptr;
  nmatrix* mat;
}nmatrix_buffer;
```

The `nmatrix_buffer` struct is used to store the data of the view of matrix. Each NMatrix view stores the pointer to the NMatrix struct of which it is a view of. Whenever a view is created of a view then also that new view would be stored as `nmatrix_buffer` and the data of that new view is created using the data from parent view and the parent NMatrix of that parent view and the NMatrix is also made parent to the newly created view. That view any number of views can be created from a matrix or a view of a matrix and so on.

```
  nmatrix_buffer* slice = ALLOC(nmatrix_buffer);
  slice->dtype = nmat->dtype;
  slice->mat = nmat;

  get_slice(nmat, lower_indices, upper_indices, slice);

  return TypedData_Wrap_Struct(NMatrix, &nm_buffer_data_type, slice);
```

Above code snippet is from the NMatrix accessor for when a slice is accessed using the `[]` method of NMatrix. For example, `X[0...2, 0...3]` would return a slice of matrix `X` which is a view of the matrix with only first 2 rows and first 3 columns of the matrix.

```
void get_slice(nmatrix* nmat, size_t* lower, size_t* upper, nmatrix_buffer* slice){
  /*
    parse the indices to form ranges for C loops
    then use them to fill up the elements
  */

  size_t slice_count = 1, slice_ndims = 0;

  for(size_t i = 0; i < nmat->ndims; ++i){
    size_t a1 = lower[i], b1 = upper[i];

    //if range len is > 1, then inc slice_ndims by 1
    //and slice_count would be prod of all ranges len
    if(b1 - a1 > 0){
      slice_ndims++;
      slice_count *= (b1 - a1 + 1);
    }
  }

  slice->count = slice_count;
  slice->ndims = slice_ndims;
  slice->shape = ALLOC_N(size_t, slice->ndims);

  size_t slice_ind = 0;
  for(size_t i = 0; i < nmat->ndims; ++i){
    size_t dim_length = (upper[i] - lower[i] + 1);
    if(dim_length == 1)
      continue;
    slice->shape[slice_ind++] = dim_length;
  }

  VALUE* state_array = ALLOC_N(VALUE, nmat->ndims);
  for(size_t i = 0; i < nmat->ndims; ++i){
    state_array[i] = SIZET2NUM(lower[i]);
  }

  // for float64
  double* nmat_elements = (double*)nmat->elements;
  size_t start_index = get_index(nmat, state_array);  // slice first element index in elements array
  slice->buffer_ele_start_ptr = (nmat_elements + start_index);
}
```

The way buffer is used in slicing is that when slicing is called, instead of copying the elements into new nmatrix struct, a `nmatrix_buffer` struct is created and just the elements pointer to used to create the pointer of buffer matrix which removes the copying of elements and hence preventing a copy.

Also, `nmatrix_buffer` points to the original matrix and uses the original shape to create strides which are used in conjunction with buffer elements starting pointer to correctly access the sliced matrix elements from the original matrix elements array. This makes the sliced matrix as a view to the original matrix and one can change the sliced matrix elements and same change can be seen in the original matrix elements.

Pull requests

https://github.com/SciRuby/ruby-sparse/pull/11

https://github.com/SciRuby/ruby-sparse/pull/14

https://github.com/SciRuby/numruby/pull/57

Work Completed

- Implemented DIA, CSR, CSC sparse matrix.
- Elementwise operations for COO, DIA, CSR and CSC. Implemented the base C function for unary and binary operators and used implemented a few operators like sin, cos, +, -, * , / using this base function. Other unary, binary operators can be created simply by specifying the operator mapping to C function or operator used for it.
- Dense to/from Sparse matrix conversion support for NumRuby NMatrix.
- Serialization support for all 4 sparse matrix implementations.
- Pretty print and inspect for all 4 sparse matrix implementations.
- Iterators for sparse matrices. `each` implemented for all 4 sparse implementations. `each_with_indices` for COO/CSR/CSC, `each_row` for CSR, `each_column` for CSC also implemented.
- Inter sparse types conversion methods. Sparse conversion among COO, CSR and CSC is implemented. DIA to/from other sparse conversion doesn't have much utility as DIA sparse has an extra constraint that only diagonal values can be non-zero, so it's not a generic sparse unlike others.

- View object addition to NumRuby.
- View integration to slicing of N-dimensional matrix.
- View integration to indexing, iteration of N-dimensional matrix.
- Replaced Data_{Get, Wrap, Make}*Struct with TypedData*(Get, Wrap, Make}_Struct in Ruby-Sparse and NumRuby.

Future work

- N-dimensional sparse matrix support.
- Linear algebra and graph modules for Ruby-Sparse.
- View integration to broadcasting of N-dimensional matrix.

Deliverables with status

- Implementation of COO, DIA, CSR and CSC sparse matrices: Completed.
- Elementwise operations (unary and binary): Core idea completed.
- Dense to/from sparse matrix conversion for all 4 sparse types: Completed.
- Serialization support for all 4 sparse types: Completed.
- Pretty print and inspect for all 4 sparse types: Completed.
- Iterators for all 4 sparse types: Completed but not all mentioned in the proposal as some of those are not logically possible, like, `each_column` for CSR.
- Inter sparse conversion: Completed but not for DIA as it is not a generic sparse type.
- Views implementation for NumRuby: Completed.
- Linear algebra support for ruby-sparse: Postponed for later.
- Multiple dtypes support for ruby-sparse: Postponed for later.
- `triu`, `tril`, `hstack` and `vstack` for sparse matrices: Postponed for later.

Conclusion

The sparse matrices are for now only implemented with just one dtype `float64` and not for `int`, `bool` etc, as simple but redundant way to do it is to use switch statement but that isn't scalable, so I decided to use ERB code templating for it but I wasn't able to figure out a way to do that for `ruby-sparse` code during the grant period so have keep for later. `RubySparse::LinAlg`, `triu`, `tril`, `hstack` and `vstack` have been postponed to be done later.

Apart from mentioned above, the core implementation all the other major features mentioned in the proposal have been implemented. Tests are not too strong yet, more tests will be added soon that are diverse and cover most of the possibilities.