Differentiating sparse matrix operations with reversible programming

Jie Li

Mentors: Jinguo Liu, Jiuning Chen

June 7, 2021

1 Summary of the Proposal

Sparse matrices are extensively used in scientific computing, however there is no automatic differentiation package in Julia yet to handle sparse matrix operations yet. This project will utilize the reversible embeded-ded domain-specific language NiLang.jl to differentiate sparse matrix operations by re-writing the sparse functions in Julia base in a reversible style. I will port the generated backward rules to ChainRules.jl as an extension, where ChainRules.jl is the most popular Julia package providing backward rules for automatic differentiation packages.

2 Background

Matrices having only a small percentage of nonzero elements are said to be sparse. Sparse matrices have been applied in many areas: e.g., linear programming, network theory, structural analyses and so on. Recently, the automatic differentiation (AD) on sparse matrices has become a popular topic in the scientific computing field. Modern AD packages depoly a broad range of computational techniques to improve applicability, run time, and memory management. Most of the well-known AD packages in the market, such as Tensorflow, Pytorch and Flux implement reverse mode AD and checkpointing techniques at the tensor level to meet the need in deep learning.

However, sometimes scientists and engineers have to define backwards rules manually for specific scientific problems, such as Hamiltonian engineering, phase transition problem and so on. Instead of defining backward rules manually, one can also use a general purposed AD (GP-AD) framework like Tapenade, Zygote and NiLang[2]. NiLang, a reversible domain-specific language, shows its great potential in generating backward rules for sparse matrices. In this project, we will implement AD for sparse matrix operations by rewriting programs in Julia base by NiLang. Furthermore, backward rules generated by NiLang will be ported to ChainRules.jl as an extension.

3 Goal and Objectives

The main target of this project is providing AD for sparse matrices by NiLang. It could be broken into three concrete goals as following:

- Implement sparse matrix operations in Julia writen by NiLang.
- Generate chain rules and export them into ChainRules.jl.
- Release an open source Julia package on AD for sparse matrices. Test converage should be above 80% and a quick start tutorial for the package should be presented.

4 Methods

In this section, we will give a detailed introduction to design and techniques in this project. We will talk about the data structure of sparse arrays in Julia first. Then we will propose how to construct AD for sparse matrices from low level operations to high level operations. Finally we consider about exporting chain rules generated by reversible functions into ChainRules.jl.

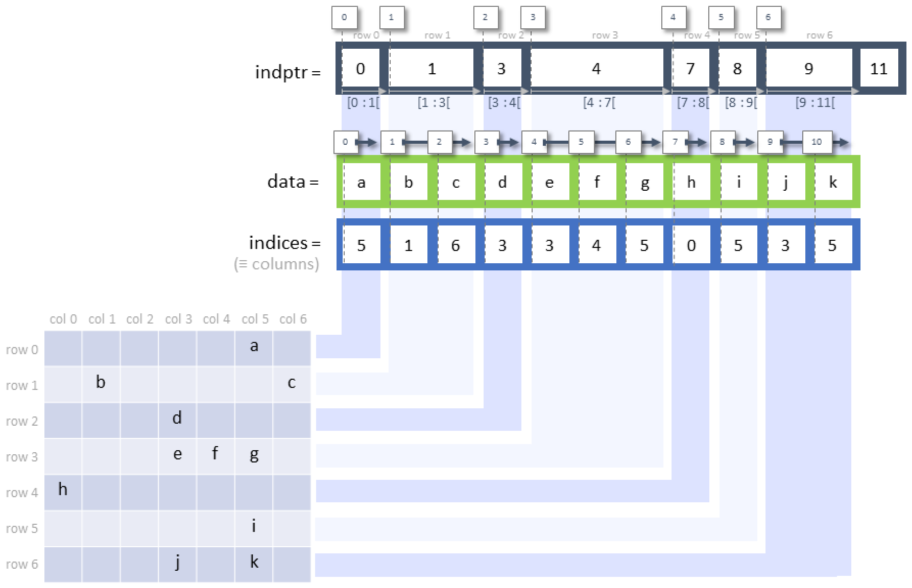


Figure 1: SparseCSR Format Illustration

4.1 SparseCSC

Julia has support for sparse vectors and sparse matrices in the SparseArrays stdlib module. As mentioned before, sparse arrays are arrays that contain enough zeros. To take advantage of this special property, sparse arrays are stored in a special data structure, known as Compressed Sparse Column (CSC) format, which leads to great savings in space and execution time compared to dense arrays.

The internal representation of SparseMatrixCSC in Julia is as follows:

```
struct SparseMatrixCSC{Tv,Ti<:Integer} <: AbstractSparseMatrixCSC{Tv,Ti}
m::Int                   # Number of rows
n::Int                   # Number of columns
colptr::Vector{Ti}       # Column j is in colptr[j]:(colptr[j+1]−1)
rowval::Vector{Ti}       # Row indices of stored values
nzval::Vector{Tv}        # Stored values, typically nonzeros
end

```

In this project, we will fully exploit the SparseMatrixCSC format to improve the performance of sparse matrix operations, e.g. matrix multiplication. Elements in columns of sparse matrix would be accessed more often whereas elements in rows of sparse would be accessed less to avoid slow operations. I mentioned and analyzed this performance gap in my blog .

4.2 Low Level Operations

Low level operations are reffered as basic matrix operations in matrix computation, e.g. BLAS. SparseArrays module in Julia provides a large amount of low level operations in linalg.jl. It could be summarized into several categories as following:

```
function mul!(C::StridedVecOrMat, A::AbstractSparseMatrixCSC, B::DenseInputVecOrMat, ::Number, ::
  Number)
function dot(A::AbstractSparseMatrixCSC{T1,S1},B::AbstractSparseMatrixCSC{T2,S2}) where {T1,T2,S1,
  S2}
function kron(C::SparseMatrixCSC,A::AbstractSparseMatrixCSC,B::AbstractSparseMatrixCSC)
function norm(A::AbstractSparseMatrixCSC, p::Real=2)
```

Low level operations above would be implemented by NiLang in the development plan. It should be addressed that low level operations are the basis in this project since it will not only provide interface to users directly but also provide support for high level operation.

4.3 High Level Operations

High level operations are reffered as operations exploiting the structure of a matrix, e.g. matrix decomposition. Matrix decomposition plays an important role in matrix computation, one of the most famous matrix decomposition is singular value decomposition (SVD).

However, SVD causes great computation when the scale of matrix is tremendous. Instead of implementing classical SVD for sparse matrices by NiLang, we would implement low rank SVD and linear Principal Component Analysis (PCA) for them. Such probabilistic algorithms would project the original sparse matrix $\mathbf{A}$ into a smaller matrix $\mathbf{B}$ and perform SVD on $\mathbf{B}$ so as to save computation.

In order to make familar with the project, I implemented the low rank SVD algorithm in Lowranksvd.jl. It served as an important reference for the implementation by NiLang.

4.4 Export Chain Rules into ChainRules.jl

We would export chain rules into ChainRules.jl when we accomplished the task of rewriting sparse matrix operations by NiLang. The delicate design of NiLang makes it easy to generate the back propogation process and gradient by calling $\sim$ *reversible_function*.

We could obtain the gradient of norm function by NiLang as following[1]:

```julia
    using NiLang, NiLang.AD, Zygote, ChainRules
    using BenchmarkTools
    # Julia native implementation of 'norm2' function.
    function norm2(x::AbstractArray{T}) where T
        out = zero(T)
        for i=1:length(x)
            @inbounds out += x[i]^2
        end
        return out
    end
    # Then we have the reversible implementation
    @i function r_norm2(out::T, x::AbstractArray{T}) where T
        for i=1:length(x)
            @inbounds out += x[i]^2
        end
    end
    x = randn(1000);
    @benchmark (~r_norm2)(GVar($(norm2(x)), 1.0), $(GVar(x))) seconds=1
```

We could generate chain rules in a reverse mode by binding the reversible function to ChainRules.rrule and export it into ChainRules.jl as following

```julia
norm2_faster(x) = norm2(x)
function ChainRules.rrule(::typeof(norm2_faster), x::AbstractArray{T}) where T
    out = norm2_faster(x)
    function pullback(y)
        ChainRules.NoTangent(), grad((~r_norm2)(GVar(out, y), GVar(x))[2])
    end
    out, pullback
end

original_grad = norm2'(x)
@assert norm2_faster'(x)  original_grad
```

5 Schedule and Timeline

This project will be shipped by four (mostly) sequential stages:

[1] `https://github.com/GiggleLiu/NiLang.jl/blob/master/examples/port_chainrules.jl`

(1) Implement low level operations by NiLang.

(2) Implement high level operations by NiLang.

(3) Export chain rules into ChainRules.jl.

(4) Release the project. Add more documentation and tutorials.

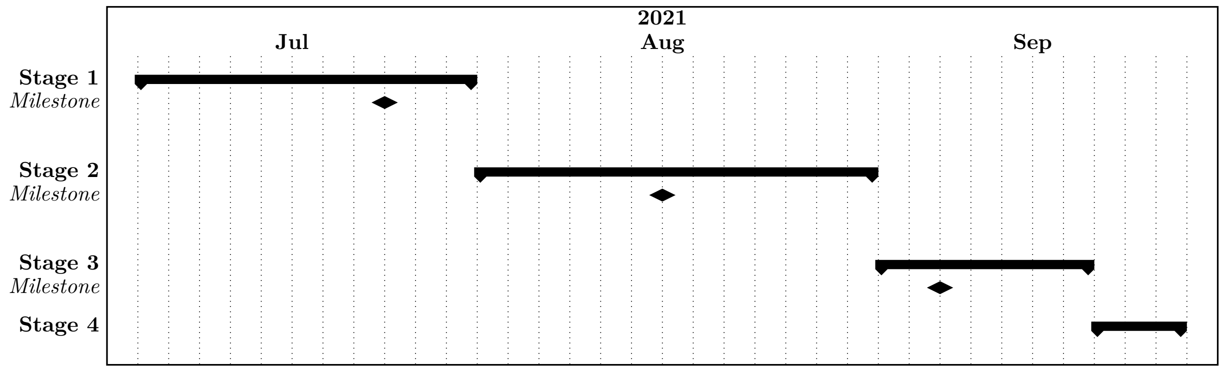
For the purpose of OSPP phase evaluation, Ill list the expected timeline of this project in the form of Gantt chart [1]

(1) 1st Jul - 31st Jul Implement low level operations by NiLang. Carefully test by CI and export chain rules into ChainRules.jl.

(2) 1st Aug - 30th Aug Implement high level operations by NiLang. Carefully test and export chain rules into ChainRules.jl.

(3) 1st Sep - 20th Sep Release the project. Add more documentation and tutorials.

(4) 21st Sep - 30th Sep Leave the time as flexible week.



OSPP 2021 has two evaluation phases, this is a reference criterion that I have in mind; whether it is adopted depends on mentors own judgement:

- Phase 1 Student and mentor should submit the phase 1 evaluation before 15th Augst. As long as the implementation of low level operations have been accomplished and passed CI building, it should be considered sufficient to pass.
- Phase 2 Student and mentor should submit the final phase evaluation before 30th September . It should be considered a pass once 1. all sparse matrix operations in the development plan have been implemented with test converage above 80% 2. there are specific documentation, tutorials.

References

[1] HL Gantt. Work, wages and profit, published by the engineering magazine. *New York*, 1910.

[2] Jin-Guo Liu and Taine Zhao. Differentiate everything with a reversible embeded domain-specific language. *arXiv preprint arXiv:2003.04617*, 2020.

Jie Li

Github: https://github.com/jieli-matrix
Website: https://jieli-matrix.github.io/
Email: li_j20@fudan.edu.cn

EDUCATION

9.2020 - 6.2023	**Master of Applied Mathematics** at Fudan University *Supervised by Young PI Weiyang Ding* *Focus on Numerical Optimization and Matrix Computation*
9.2016 - 7.2020	**Bachelor of Mathematics and Applied Mathematics** at Lanzhou University

SKILLS AND QUALIFICATIONS

Programming Languages

Advanced skills	Python, Julia, Matlab
Basic skills	Git, Linux, Cpp

Languages

Native	Chinese
Advanced	English (TOEFL:94 CET-4:632 CET-6:571)

INTERNSHIP EXPERIENCE

10.2019 - 12.2019 **NLP Researcher in Core Development Platform** at iFLYTEK

- Automatic Pipeline on Senior High School Math Homework
- Code Implementation of Multi-Label Learning with Deep Forest

PROJECTS

5.2021 - now **Lowranksvd.jl** Lowranksvd.jl

- Algorithm 4.4 from Halko et al. implemented by Julia with all CI tests passing.
- Algorithm 5.1 from Halko et al. implemented by Julia with all CI tests passing.

4.2020 **CPUPredict** CPUPredict

Time series prediction of the CPU's usage rates.
This project won the third place in Elastic Cloud College Challenge held by ALibaba.

CONTESTS

11.2020 **Brain-Inspired Intelligence Contests held by Fudan University**

Efficent information encoding and energy efficiency in E/I balanced neuronal networks

Numerical Simulation for E/I balanced neuronal networks.
I won the third place in the contest.