Differentiating sparse matrix operations with reversible programming

Jie Li

Mentors: Jinguo Liu, Jiuning Chen

September 30, 2021

1 Project Information

1.1 Scheme Description

Sparse matrices are extensively used in scientific computing, however there is no automatic differentiation package in Julia yet to handle sparse matrix operations. This project utilizes the reversible embedded domain-specific language NiLang.jl to differentiate sparse matrix operations by writing the sparse matrix operations in a reversible style. The generated backward rules are ported to ChainRules.jl as an extension, so that one can access these features in an automatic differentiation package like Zygote, Flux and Diffractor directly.

1.2 Time Planning

This project is shipped by four (mostly) sequential stages:

(1) Implement low level operations by NiLang.

(2) Carefully test by CI and export chain rules into ChainRules.jl.

(3) Implement high level operations and perform AD operations.

(4) Add some use cases and enhance docs.

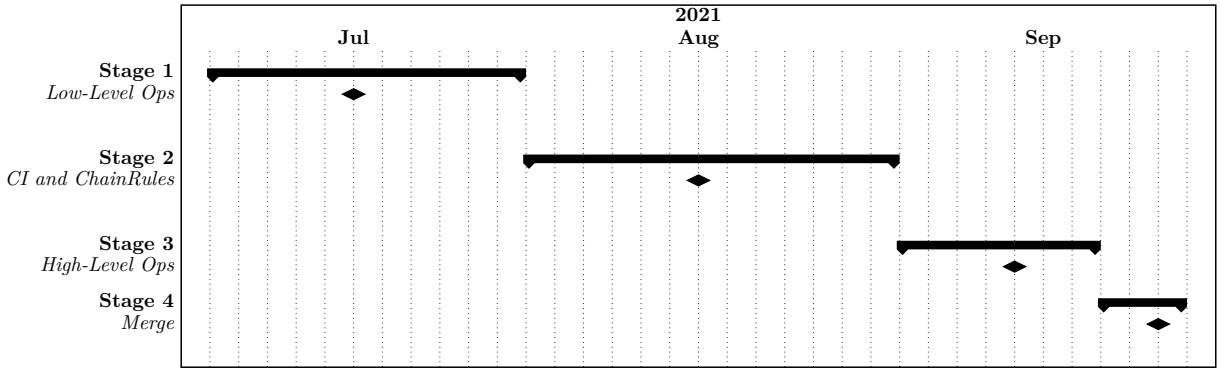
I list the timeline of this project in the form of Gantt chart.

(1) 1st Jul - 31st Jul Implement low level operations by NiLang.

(2) 1st Aug - 15th Aug Carefully test by CI and export chain rules into ChainRules.jl.

(3) 15th Sep - 20th Sep Implement high level operations and perform AD operations.

(4) 21st Sep - 30th Sep Add some use cases and enhance docs.



2 Project Summary

2.1 Project Output

(1) Differentiate sparse matrix operations in Julia base by rewriting the sparse functions in NiLang.jl.
Status: *Already completed. Black-box tests on NiLang programs have passed in CI.* Related PR

(2) Port the generated backward rules to ChainRules.jl as an extension.
Status: *Already completed. Backward rules have been checked by ChainRulesTestUtils.* Related PR

(3) Release an open source julia package with test coverage over 85%.
Status: *Test coverage achieved at 87%.* Code Converage Report

(4) Add some use cases for getting a start.
Status: *Already completed. A simple use case is shown in ReadMe.* Related PR

2.1.1 Demo

In this section, an eigen-solver case and a low rank SVD case are illustrated to show how to use the package.

Eigen-solver case Here, I want to calculate the eigenvector $\vec{x}$ corresponding to the maximum eigenvalue of matrix **A**.

```
    # find max eigenvalue by power method
function power_max_eigen(A, x, target; niter=100)
    for i=1:niter
        x = A * x
        x /= norm(x)
    end
    return abs(x' * target)
end
```

I use the similarity between $\vec{x}$ and the target vector as loss function, which would adjust $\vec{x}$ to the expected one. Since I have exported rules on sparse matrix into ChainRules, I could calculate gradient of power_max_eigen simply as follows.

```
    using Zygote
    using SparseArrays, LinearAlgebra
    using BenchmarkTools
    # generate test data
    A = sprand(5000, 5000, 0.1)
    x = randn(5000)
    target = randn(5000)

    # Zygote is able to generate correct gradient for sparse matrix and dense vectors
    # in computation form of dense arrays
    @btime max_ga_z, max_gx_z, max_gt_z = Zygote.gradient(power_max_eigen, $A, $x, $target) #12.954
      s (4272 allocations: 27.91 GiB)
    using SparseArraysAD
    # The gradient generated by SparseArraysAD is much faster
    # since it keeps the orginal type in computation process
    @btime max_ga, max_gx, max_gt = Zygote.gradient(power_max_eigen, $A, $x, $target) # 6.180 s (5072
      allocations: 16.75 GiB)
```

Here I compared the memory allocations and computation speed between our package and Zygote. You will see that using SparseArraysAD would not only speed up the computation process but also save much memory since our implementation does not convert a sparse matrix to a dense arrays in gradient computation.

Low Rank SVD case One of the most important steps in low_rank_algorithm is to calculate the approximate basis, which invloves with matrix multiplication and QR decomposition. I ensembled QR rules in the package since official QR rules has not been included in ChainRules yet. SparseArraysAD could speed up the solution since I have exported rules of sparse matrix multiplication.

```julia
function svd_loss(A::AbstractSparseMatrix{T}, z::Vector{T}, r::Int) where T
    U, S, Vt = low_rank_svd(A, r)
    residual_mat = U * Diagonal(S) * Vt - U * Diagonal(z) * Vt
    return tr(residual_mat'*residual_mat) # square of Frobenius norm
end

# generate test data
r = 10
A = sprand(100, r, 0.2) * Diagonal(rand(r)) * sprand(r, 50, 0.2)
z = rand(r)

grad_A = Zygote.gradient(A -> svd_loss(A, z, r), A)[1]
grad_fA = FiniteDifferences.grad(FiniteDifferences.central_fdm(5, 1), A -> svd_loss(A, z, r), A)[1]
```

I use FiniteDifferences to check the results computed by SparseArraysAD and all checks passed.

2.2 Scheme Progress

1. 1st Jul - 31st Jul Implement low level operations by NiLang.
   I have rewritten almost all sparse matrix multiplication and dot operations in Julia base by NiLang. I list the implemented sparse matrix operators as follows. All the functions listed have passed CI tests and the code coverage have achieved over 85%.

```julia
 function imul!(C::StridedVecOrMat, A::AbstractSparseMatrix{T}, B::DenseInputVecOrMat, ::Number, ::Number) where T
 function imul!(C::StridedVecOrMat, xA::Adjoint{T, <:AbstractSparseMatrix}, B::DenseInputVecOrMat, ::Number, ::Number) where T
 function imul!(C::StridedVecOrMat, X::DenseMatrixUnion, A::AbstractSparseMatrix{T}, ::Number, ::Number) where T
 function imul!(C::StridedVecOrMat, X::Adjoint{T1, <:DenseMatrixUnion}, A::AbstractSparseMatrix{T2}, ::Number, ::Number) where {T1, T2}
 function imul!(C::StridedVecOrMat, X::DenseMatrixUnion, xA::Adjoint{T, <:AbstractSparseMatrix}, ::Number, ::Number) where T
 function idot(r, A::SparseMatrixCSC{T},B::SparseMatrixCSC{T}) where {T}
 function idot(r, x::AbstractVector, A::AbstractSparseMatrix{T1}, y::AbstractVector{T2}) where {T1, T2}
 function idot(r, x::SparseVector, A::AbstractSparseMatrix{T1}, y::SparseVector{T2}) where {T1, T2}

```

2. 1st Aug - 15th Aug Carefully test by CI and export chain rules into ChainRules.jl.
   I have exported the backward rules into ChainRules.jl. I used ChainRulesTestUtils.jl to test correctness and robustness of sparse matrix AD rules. All the rules have passed CI tests.

3. 15th Sep - 20th Sep Implement high level operations and perform AD operation.
   I have implemented low rank svd algorithm in Julia. To perform AD on low_rank_svd algorithm, I wrapped backward rules of QR decomposition since QR rules haven't been ensembled in ChainRules. I used finite differences method to check the correctness of the algorithm. All the code have passed CI tests and the code converage have achieved at 87% finally.

4. 21st Sep - 30th Sep Add some use cases and enhance docs.
   A simple use case has been added into ReadMe in the project.

2.3 Problems and Solutions

To complete the projcet, one should be equipped with both good sense of numerical linear algebra and automatic differentiation **in theory** and familiarity with NiLang.jl, ChainRules.jl and so on **in practice**. It took me much of time to figure out how automatic differentiation works out in computation process. In this section, I would only introduce the problems and solutions I came cross in practice.

2.3.1 Incorrect gradient of sparse vector

When I tried to export chain rules of function *dot*, all the checks failed in ChainRulesTestUtils. Then, I printed the gradient calculated by NiLang and found it proved to be zero which was really abnormal. I fed back the strange phenomenon to mentor and he told me that iterator can not carry gradients and the dangerous feature would be removed from NiLang in the future. After I corrected the codes according to his suggestions, all the checks passed. The problem also gave a good reason to remove iterator support to avoid potential missing of gradients in NiLang, which had been metioned in this issue.

2.3.2 Test on Generated Rules

In our project, I calculated the gradient of sparse matrix in reversal mode in NiLang. To test the correctness of the gradient, I should calculate gradient in an another way. There are many choices such as FiniteDifferences, ForwardDiff and so on. After discussing with mentors, I decided to use ChainRulesTestUtils to check the correctness of the gradient, which was highly recommended by Lyndon White in JuliaCon2020. At first, I implemented tangent of sparse matrix by myself, it is a bit hard task for me since there are no related examples. Later, mentor and I talked about this problem with Lyndon White, the core founder of ChainRules, in slack and he mentioned rand_tangent method may help. I read related source code in CRTU and made some changes so as to keep the adjoint gradient could keep the sparsity. Finally all the rules checked by CRTU and passed all the tests successfully.

2.3.3 Add support for QR rules

The low_rank_svd involved with QR decomposition which was used to find the approximate basis. However, backward rules of QR decomposition were ensembled neither in ChainRules nor Zygote. In order to calculate the gradient of low_rank_svd, I wrapped the QR rules in ChainRules and checked by CRTU in help of mentor. I would like to enhance the implementation of QR rules (keep Q factor and R factor in compactWY format) and pull request to ChainRules.jl in the future.

2.4 Development Quality

1. Code style
   Blue style was recommended in Julia. Since I was a freshman in Julia, I read related code and initiated their writing style. I was really amazing at the delicate code structure when I read source code in SparseArrays, FiniteDifferences and ChainRulesTestUtils. I tried to initiate such projects and made code in the project more readable and clearer.

2. Project Structure
   I used the PkgTemplate in Julia, which was the standard project template in Julia. Core source code and auxiliary test code were splited in different folders. It was easy to perform unit tests and add or remove dependencies for developers. Besides, it was also convenient for users to add the package and use it.

3. Robustness
   I wrote test code for each function in source code and code coverage achieved at 87% finally. Furthermore, Continuous Integration (CI) has been set in Github Actions, which would be triggered automatically when I commited code to github repository. Updated changes would be merged into master branch only when all the checks passed in CI.

4. Work Pipeline
   I really appreciated great efforts of mentors. When I came across abnormal bugs, they would discuss the problems with me and provide useful suggestions. After the code passed CI tests, they would review my code carefully and then approved or merged the updates into master branch.

2.5 Communication and Feedback with Mentor

Community mentors Jinguo and Jiuning kept track of the project and guided me patiently during the whole process. Routine meetings were held to talk development plan and we debug the broken programs by living coding and discussion on github. They gave insightful feedback to me which would help me to improve myself in the future listed as follows.

- Learn how to write a good issue: declare the problems clearly and don't let others to guess your problems from logs.

- Communicate with other developers in the community and don't hesitate to ask questions.
- Keep an open mind to other AD packages and use even improve them.