Differentiating sparse matrix operations with reversible programming

Jie Li

Mentors: Jinguo Liu, Jiuning Chen

August 14, 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 yet. This project will utilize the reversible embedded 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.

1.2 Time Planning

This project will be 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 by NiLang and export chain rules.

(4) Rebase the project on NiLang.Core and merge it into NiLang.jl.

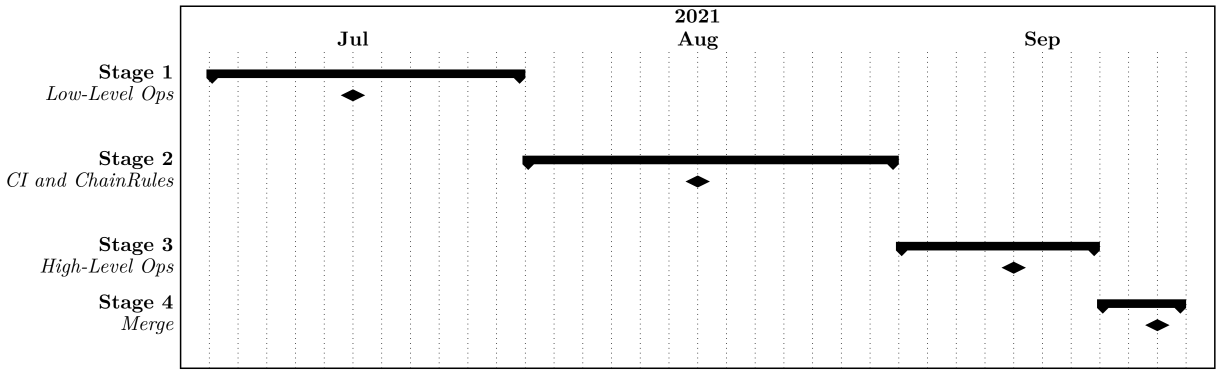
I list the timeline of this project in the form of Gantt chart.

(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 by NiLang and export chain rules.

(4) 21st Sep - 30th Sep Rebase the project on NiLang.Core and merge it into NiLang.jl.



2 Project Schedule

2.1 The Accomplished Work

In the first phase, we unified the sparse matrix operations and established the whole pipeline for automatic differentiation in the project.

Unify Sparse Matrix Operations Initial goal of the project is to implement sparse matrix operators and generate corresponding automatic differentiation compared with Pytorch. However, as mentioned in Issue in Pytorch, the users have difficulties in choosing the matmul function they need since the functionalities of these functions is not independent to each other. I outlined the sparse matrix operators to be implemented as follows.

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

It is considered to be a suitable unification since the protype of *imul!* is gaxpy operation in numerical maths, which is a generalization to matrix multiplication. Besides, Julia's type system and dot operations are also taken into account. So far, seven of the methods have been implemented and all tests have passed (both in local and CI). For more detailed implementations and test cases, check for this branch and the test converage is over 80%.

Generate Automatic Differentiation Rules Ultimate goal of the project is to provide Automatic Differentiation (AD) rules on sparse matrix operators generated by NiLang. To outline the development in the AD section, one should consider how to generate rules and how to test the accuracy of rules. There are many choices to do these and I finally establish the pipeline for AD part in this project as follows.

(1) Write rrule in support of ChainRulesCore.jl. Rules would be generated by NiLang.

(2) Test rrule in support of ChainRulesTestUtils.jl.

(3) Provide the passed rules to ChianRules.jl.

The pipeline is also consistent with the idea metioned in JuliaCon 2020 by Lyndon White. So far, I generate AD rules by NiLang and ChainRulesCore and test rules by ChainRulesTestUtils under the help of mentors. For more detailed implementations and test cases, check for this branch and the test converage is over 80%.

2.2 Problem and Solution

It is the most difficult project for me so far since I am totally new to NiLang and AD. For NiLang, it would be easy for me to write the code in syntax but the high performance couldn't be ensured. For AD, I even couldn't figure out how AD works in computation theory before but I need to generate AD rules in this project.

2.2.1 Problems on NiLang Section

Performance Gap In benchmark tests, we found the implementation of matrix multiplication between Adjoint sparse matrix and dense matrix is about 2x slower than the one in Julia base. Finally, we found the reason by profiling the program.

```
julia> @profile for i=1:100 imul_v1!(C, A', b, 1.0, 1.0) end

julia> Profile.print(mincount=20; format=:flat);
 Count  Overhead File                                    Line Function
 =====  ======== ====                                    ==== ========
  1900         0 @Base/Base.jl                             39 eval
  1900         0 REPL[42]                                   1 macro expansion
   217       217 @Base/array.jl                           802 getindex
  1126      1126 @Base/array.jl                           841 setindex!
  1900         0 @Base/boot.jl                            360 eval
  1900         0 @Base/essentials.jl                      708 #invokelatest#2
  1900         0 @Base/essentials.jl                      706 invokelatest(::Any)
    30        30 @Base/float.jl                           332 *
   242       242 @Base/float.jl                           326 +
  1900         0 @Base/logging.jl                         603 with_logger
  1900         0 @Base/logging.jl                         491 with_logstate(f::Function, logstate::Any)
   255       255 @Base/promotion.jl                       410 ==
   263         8 @Base/range.jl                           674 iterate
  1900         0 @Base/task.jl                            411 (::VSCodeServer.var"#53#54")()
   272         0 @NiLangCore/src/Core.jl                  232 #_#36
   272         0 @NiLangCore/src/Core.jl                  232 PlusEq
   263         0 @NiSparseArrays/src/linalg.jl             47 imul_v1!(C::Matrix{Float64}, xA::LinearAlgebra.Adjoint{
      Float64, SparseMatrixCSC{Float64, Int64}}, B::Matrix{Fl...
   247         0 @NiSparseArrays/src/linalg.jl             48 imul_v1!(C::Matrix{Float64}, xA::LinearAlgebra.Adjoint{
      Float64, SparseMatrixCSC{Float64, Int64}}, B::Matrix{Fl...
  1375         0 @NiSparseArrays/src/linalg.jl             50 imul_v1!(C::Matrix{Float64}, xA::LinearAlgebra.Adjoint{
      Float64, SparseMatrixCSC{Float64, Int64}}, B::Matrix{Fl...
  1900         0 @VSCodeServer/src/eval.jl                 34 macro expansion
  1900         0 @VSCodeServer/src/repl.jl                124 (::VSCodeServer.var"#68#70"{Module, Expr, REPL.
      LineEditREPL, REPL.LineEdit.Prompt})()
  1900         0 @VSCodeServer/src/repl.jl                123 (::VSCodeServer.var"#69#71"{Module, Expr, REPL.
      LineEditREPL, REPL.LineEdit.Prompt})()
  1900         0 @VSCodeServer/src/repl.jl                157 repleval(m::Module, code::Expr, #unused#::String)
  1900         0 @Profile/src/Profile.jl                   28 top-level scope
Total snapshots: 3800
```

The setindex operation takes to much time. It is related to "trait" of NiLang that it assigns an input variable back to the array. If we copy the element out first to a scalar, it would ease the problem. After correcting the problem, the performance gap is constrained into 1.2 times. For more detailed dicussion on this problem, check for this PR.

Gustavson Sparse Matrix Multiplication Different from traditional gaxp operations, Gustavson algorithm is applied in sparse matrix multiplication. One of the high efficiency comes from the estimate on the number of non-zero elements in **C**. However, it is hard to define the reversible operations on the extension and shrink of the memory in NiLang. We pended the problem in this issue now and left it for further exploration.

2.2.2 Problems on AD Section

Establish the whole pipeline As I mentioned before, one should consider how to generate rules and how to test the accuracy of rules. In fact, I completed the implementation of sparse matrix operators much earlier before August. However, I am new to AD both in theory and techniques and spend much time on AD part. At first, I wrote the tests on calculating the jacobian matrix using ForwardDiff. I realized I didn't care the jacobian matrix but rrule and the tests on rrule until this week. However, I still don't know to choose which package to test the accuracy of rules in Julia since there are many choices provided in scalar cases in ChainRules.jl. I'd like to thank mentors for pointing the problem and recommending ChainRulesCore / ChainRulesTestUtils to do this.

How to complete an unfamiliar project in a short time In this part, I'd like to mention a general problem students involed in OSPP projects would face: Students learn the basics or maybe the outdated knowledge in school but they would try to complete the project using the techniques / theory they never heard before. Take myself experience as an example. If one wants to learn about a new feild, it usually begins with a classical tutorial or a friendly course. However, I don't have so much time to learn AD since OSPP is a three-month project not a course. I read papers about ForwardDiff and Tapenade recommended by mentors; I read blogs and watched open talks on AD in Machine Learning and Scientific Computation; I watched the AD talk on JuliaCon times by times since people commented it is not newcomer-friendly. Now I could summarize the experience as follows: *Differentiation is originally a pure mathematical concept, and we could make it Automatic by Tangent Mode and Adjoint Mode. Then AD is extended to the computaion concept, we want to calculate the differentiation of a program. Many techniques have been proposed such as operator overloading, source code transformation and so on. Julia have many AD packages under development like Zygote, ForwardDiff, etc. It would cost one much time to consider the relationship between the AD theory and packages but it would also be exciting.* So what could we learn from the example? We shouldn't expect the candidate to be the related expert or researcher. The open source community could release more good first issues and draw a more systematic introduction to the framework and techniques in the project. The OSPP students are also encouraged to share their experience and help newcomers to join in the community.

2.3 Subsequent Work Arrangement

In the subsequent, we would focus on the four main parts:

(1) Implementation of high-level operators.

(2) Generate rrules and test the accuracy for all operators.

(3) Rebase the project on NiLang.Core and merge it into NiLang.

(4) Explore the possible solutions for pending problems like Gustavson algorithm.