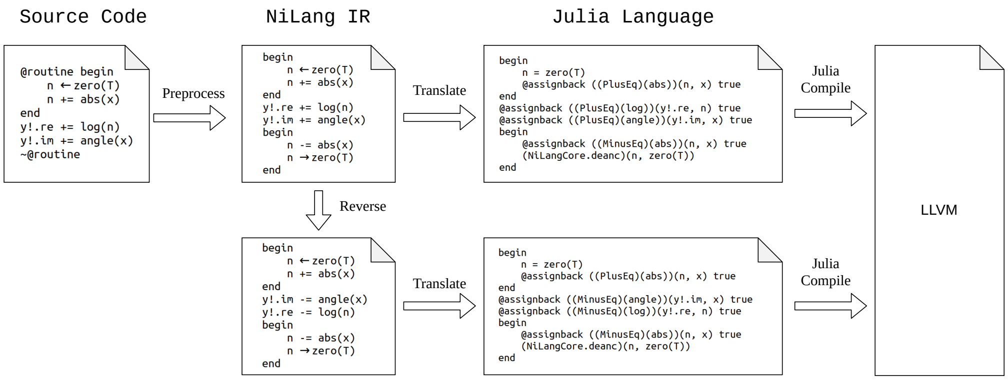


Figure 1: Compiling the body of the complex valued log function defined in Listing. **??**.

Differentiate Everything with a Reversible Domain-Specific Language: supplimentary materials

Anonymous Author(s)
Affiliation
Address
`email`

In Sec. 1.2, we introduce the detailed design of NiLang, including the grammar, the instruction set and the contructor. In Sec. 2, we show the example codes used in the benchmarks. In Sec. 3, we discuss some important issues, the space time tradeoff, the instruction set, the gradient on ancilla problem, the shared memory problem, and also the future directions to go.

1 NiLang in Detail

1.1 NiLang Compilation

The compilation of a reversible function to native Julia functions is consisted of three stages: *preprocessing*, *reversing* and *translation* as shown in Fig. 1.

In the *preprocessing* stage, the compiler pre-processes human inputs to the reversible NiLang IR. The preprocessor removes redundant grammars and expands shortcuts. In the left most code box in Fig. 1, one uses `@routine <stmt>` statement to record a statement, and `~@routine` to insert the corresponding inverse statement for uncomputing. The computing-uncomputing macros `@routine` and `~@routine` is expanded in this stage. In the *reversing* stage, based on this symmetric and reversible IR, the compiler generates reversed statements. In the *translation* stage, the compiler translates this reversible IR as well as its inverse to native Julia code. It adds `@assignback` before each function call, inserts codes for reversibility check, and handle control flows. As a final step, the compiler attaches a return statement that returns all updated input arguments at the end of a function definition. Now, the function is ready to execute on the host language.

Submitted to 34th Conference on Neural Information Processing Systems (NeurIPS 2020). Do not distribute.

1.2 NiLang Grammar

To define a reversible function one can use “@i” plus a standard function definition like bellow

```julia
"""
docstring...
"""
@i function f(args..., kwargs...) where {...}
    <stmts>
end
```

where the definition of “<stmts>” are shown in the grammar page bellow. The following is a list of terminologies used in the definition of grammar

- *ident*, symbols
- *num*, numbers
- ϵ, empty statement
- *JuliaExpr*, native Julia expression
- [ ], zero or one repetitions.

Here, all *JuliaExpr* should be pure. Otherwise, the reversibility is not guaranteed. Dataview is a view of data. It can be a bijective mapping of an object, an item of an array, or a field of an object.

$$
\begin{array}{rcl}
\langle\text{Stmts}\rangle & ::= & \epsilon \\
 & & \mid\ \langle\text{Stmt}\rangle \\
 & & \mid\ \langle\text{Stmts}\rangle\ \langle\text{Stmt}\rangle \\
\langle\text{Stmt}\rangle & ::= & \langle\text{BlockStmt}\rangle \\
 & & \mid\ \langle\text{IfStmt}\rangle \\
 & & \mid\ \langle\text{WhileStmt}\rangle \\
 & & \mid\ \langle\text{ForStmt}\rangle \\
 & & \mid\ \langle\text{InstrStmt}\rangle \\
 & & \mid\ \langle\text{RevStmt}\rangle \\
 & & \mid\ \langle\text{AncillaStmt}\rangle \\
 & & \mid\ \langle\text{TypecastStmt}\rangle \\
 & & \mid\ \langle\text{@routine}\rangle\ \langle\text{Stmt}\rangle \\
 & & \mid\ \langle\text{@safe}\rangle\ \textit{JuliaExpr} \\
 & & \mid\ \langle\text{CallStmt}\rangle \\
\langle\text{BlockStmt}\rangle & ::= & \textit{begin}\ \langle\text{Stmts}\rangle\ \textit{end} \\
\langle\text{RevCond}\rangle & ::= & \textit{(}\ \textit{JuliaExpr}\ \textit{,}\ \textit{JuliaExpr}\ \textit{)} \\
\langle\text{IfStmt}\rangle & ::= & \textit{if}\ \langle\text{RevCond}\rangle\ \langle\text{Stmts}\rangle\ [\textit{else}\ \langle\text{Stmts}\rangle]\ \textit{end} \\
\langle\text{WhileStmt}\rangle & ::= & \textit{while}\ \langle\text{RevCond}\rangle\ \langle\text{Stmts}\rangle\ \textit{end} \\
\langle\text{Range}\rangle & ::= & \textit{JuliaExpr}\ \textit{:}\ \textit{JuliaExpr}\ [\textit{:}\ \textit{JuliaExpr}] \\
\langle\text{ForStmt}\rangle & ::= & \textit{for}\ \textit{ident}\ =\ \langle\text{Range}\rangle\ \langle\text{Stmts}\rangle\ \textit{end} \\
\langle\text{KwArg}\rangle & ::= & \textit{ident}\ =\ \textit{JuliaExpr} \\
\langle\text{KwArgs}\rangle & ::= & [\langle\text{KwArgs}\rangle\ ,]\ \langle\text{KwArg}\rangle \\
\langle\text{CallStmt}\rangle & ::= & \textit{JuliaExpr}\ \textit{(}\ [\langle\text{DataViews}\rangle]\ [\textit{;}\ \langle\text{KwArgs}\rangle]\ \textit{)} \\
\langle\text{Constant}\rangle & ::= & \textit{num}\ \mid\ \pi\ \mid\ \textit{true}\ \mid\ \textit{false} \\
\langle\text{InstrBinOp}\rangle & ::= & \texttt{+=}\ \mid\ \texttt{-=}\ \mid\ \veebar\!= \\
\langle\text{InstrTrailer}\rangle & ::= & [.]\ \textit{(}\ [\langle\text{DataViews}\rangle]\ \textit{)} \\
\langle\text{InstrStmt}\rangle & ::= & \langle\text{DataView}\rangle\ \langle\text{InstrBinOp}\rangle\ \textit{ident}\ [\langle\text{InstrTrailer}\rangle] \\
\langle\text{RevStmt}\rangle & ::= & \sim\ \langle\text{Stmt}\rangle \\
\langle\text{AncillaStmt}\rangle & ::= & \textit{ident}\ \leftarrow\ \textit{JuliaExpr} \\
 & & \mid\ \textit{ident}\ \rightarrow\ \textit{JuliaExpr} \\
\langle\text{TypecastStmt}\rangle & ::= & \textit{(}\ \textit{JuliaExpr}\ \texttt{=>}\ \textit{JuliaExpr}\ \textit{)}\ \textit{(}\ \textit{ident}\ \textit{)} \\
\langle\text{@routine}\rangle & ::= & \textit{@routine}\ \textit{ident}\ \langle\text{Stmt}\rangle \\
\langle\text{@safe}\rangle & ::= & \textit{@safe}\ \textit{JuliaExpr} \\
\langle\text{DataViews}\rangle & ::= & \epsilon \\
 & & \mid\ \langle\text{DataView}\rangle \\
 & & \mid\ \langle\text{DataViews}\rangle\ ,\ \langle\text{DataView}\rangle \\
 & & \mid\ \langle\text{DataViews}\rangle\ ,\ \langle\text{DataView}\rangle\ \ldots \\
\langle\text{DataView}\rangle & ::= & \langle\text{DataView}\rangle\ \textit{[}\ \textit{JuliaExpr}\ \textit{]} \\
 & & \mid\ \langle\text{DataView}\rangle\ .\ \textit{ident} \\
 & & \mid\ \textit{JuliaExpr}\ \textit{(}\ \langle\text{DataView}\rangle\ \textit{)} \\
 & & \mid\ \langle\text{DataView}\rangle\ ' \\
 & & \mid\ \text{-}\ \langle\text{DataView}\rangle \\
 & & \mid\ \langle\text{Constant}\rangle \\
 & & \mid\ \textit{ident}
\end{array}
$$

Table 1 shows the meaning of some selected statements and how they are reversed.

Statement	Meaning	Inverse
`<f>(<args>...)`	function call	`(~<f>)(<args>...)`
`<f>.(<args>...)`	broadcast a function call	`<f>.(<args>...)`
`<y> += <f>(<args>...)`	inplace add instruction	`<y> -= <f>(<args>...)`
`<y> ⊻= <f>(<args>...)`	inplace XOR instruction	`<y> ⊻= <f>(<args>...)`
`<a> ← <expr>`	allocate a new variable	`<a> → <expr>`

`begin`
`  <stmts>`
`end`
	statement block	`begin`
		`  ~(<stmts>)`
		`end`

`if (<pre>, <post>)`
`  <stmts1>`
`else`
`  <stmts2>`
`end`
	if statement	`if (<post>, <pre>)`
		`  ~(<stmts1>)`
		`else`
		`  ~(<stmts2>)`
		`end`

`while (<pre>, <post>)`
`  <stmts>`
`end`
	while statement	`while (<post>, <pre>)`
		`  ~(<stmts>)`
		`end`

`for <i>=<m>:<s>:<n>`
`  <stmts>`
`end`
	for statement	`for <i>=<m>:-<s>:<n>`
		`  ~(<stmts>)`
		`end`

Table 1: Basic statements in NiLang IR. "~" is the symbol for reversing a statement or a function. "." is the symbol for the broadcasting magic in Julia, `<pre>` stands for precondition, and `<post>` stands for postcondition "`begin <stmts> end`" is the code block statement in Julia. It can be inverted by reversing the order as well as each element in it.

1.3 Instructions Used in Main Text

A table instructions used in the main text

instruction	output
SWAP(a, b)	b, a
ROT(a, b, θ)	$a \cos\theta - b \sin\theta, b \cos\theta + a \sin\theta, \theta$
IROT(a, b, θ)	$a \cos\theta + b \sin\theta, b \cos\theta - a \sin\theta, \theta$
$y \mathrel{+}= a * b$	$y + a * b, a, b$
$y \mathrel{+}= a/b$	$y + a/b, a, b$
$y \mathrel{+}= a\hat{}b$	$y + a^b, a, b$
$y \mathrel{+}= \text{identity}(x)$	$y + x, x$
$y \mathrel{+}= \exp(x)$	$y + e^x, x$
$y \mathrel{+}= \log(x)$	$y + \log x, x$
$y \mathrel{+}= \sin(x)$	$y + \sin x, x$
$y \mathrel{+}= \cos(x)$	$y + \cos x, x$
$y \mathrel{+}= \text{abs}(x)$	$y + \lvert x \rvert, x$
NEG(y)	$-y$
INC(y)	$y + 1$
DEC(y)	$y - 1$

Table 2: Predefined reversible instructions in NiLang.

1.4 Reversible Constructors

So far, the language design is not too different from a traditional reversible language. To port Julia's type system better, we introduce dataviews. The type used in the reversible context is just a standard Julia type with an additional requirement of having reversible constructors. The inverse of a constructor is called a "destructor", which unpacks data and deallocates derived fields. A reversible constructor is implemented by reinterpreting the `new` function in Julia. Let us consider the following statement.

```
x ← new{TX, TG}(x, g)
```

The above statement is similar to allocating an ancilla, except that it deallocates `g` directly at the same time. Doing this is proper because `new` is special that its output keeps all information of its arguments. All input variables that do not appear in the output can be discarded safely. Its inverse is

```
x → new{TX, TG}(x, g)
```

It unpacks structure `x` and assigns fields to corresponding variables in the argument list. The following example shows a non-complete definition of the reversible type `GVar`.

```
julia> using NiLangCore

julia> @i struct GVar{T,GT} <: IWrapper{T}
           x::T
           g::GT
           function GVar{T,GT}(x::T, g::GT)
                        where {T,GT}
               new{T,GT}(x, g)
           end
           function GVar(x::T, g::GT)
                        where {T,GT}
               new{T,GT}(x, g)
           end
           @i function GVar(x::T) where T
               g ← zero(x)
               x ← new{T,T}(x, g)
           end
           @i function GVar(x::AbstractArray)
               GVar.(x)
           end
       end

julia> GVar(0.5)
GVar{Float64,Float64}(0.5, 0.0)

julia> (~GVar)(GVar(0.5))
0.5

julia> (~GVar)(GVar([0.5, 0.6]))
2-element Array{Float64,1}:
 0.5
 0.6
```

`GVar` has two fields that correspond to the value and gradient of a variable. Here, we put `@i` macro before both `struct` and `function` statements. The ones before functions generate forward and backward functions, while the one before `struct` moves `~GVar` functions to the outside of the type definition. Otherwise, the inverse function will be ignored by Julia compiler.

Since an operation changes data inplace in NiLang, a field of an immutable instance should also be " modifiable". Let us first consider the following example.

```
julia> arr = [GVar(3.0), GVar(1.0)]
2-element Array{GVar{Float64,Float64},1}:
 GVar{Float64,Float64}(3.0, 0.0)
 GVar{Float64,Float64}(1.0, 0.0)

julia> x, y = 1.0, 2.0
(1.0, 2.0)

julia> @instr -arr[2].g += x * y
2.0

julia> arr
2-element Array{GVar{Float64,Float64},1}:
 GVar{Float64,Float64}(3.0, 0.0)
 GVar{Float64,Float64}(1.0, -2.0)
```

In Julia language, the assign statement above will throw a syntax error because the function call "-" can not be assigned, and `GVar` is an immutable type. In NiLang, we use the macro `@assignback` to modify an immutable data directly. It translates the above statement to

```
1 res = (PlusEq(*))(-arr[2].g, x, y)
2 arr[2] = chfield(arr[2], Val(:g),
3     chfield(arr[2].g, -, res[1]))
4 x = res[2]
5 y = res[3]
```

The first line `PlusEq(*)(-arr[3].g, x, y)` computes the output as a tuple of length 3. At lines 2-3, `chfield(x, Val{:g}, val)` modifies the `g` field of `x` and `chfield(x, -, res[1])` returns `-res[1]`. Here, modifying a field requires the default constructor of a type not overwritten. The assignments in lines 4 and 5 are straightforward. We call a bijection of a field of an object a "dataview" of this object, and it is directly modifiable in NiLang. The definition of dataview can be found in Appendix 1.2.

2 Examples

In this section, we introduce several examples.

- sparse matrix dot product,
- first kind bessel function and memory oriented computational graph,
- solving the graph embedding problem.

All codes for this section and the next benchmark section are available in the paper repository.

2.1 Sparse Matrices

Differentiating sparse matrices is useful in many applications, however, it can not benefit directly from generic backward rules for the dense matrix because the generic rules do not keep the sparse structure. In the following, we will show how to convert a irreversible Frobenius dot product code to a reversible one to differentiate it. Here, the Frobenius dot product is defined as `trace(A'B)`. In SparseArrays code base, it is implemented as follows.

```
function dot(A::AbstractSparseMatrixCSC{T1,S1},
     B::AbstractSparseMatrixCSC{T2,S2}
     ) where {T1,T2,S1,S2}
  m, n = size(A)
  size(B) == (m,n) || throw(DimensionMismatch("
     matrices must have the same dimensions"))
  r = dot(zero(T1), zero(T2))
  @inbounds for j = 1:n
    ia = getcolptr(A)[j]
    ia_nxt = getcolptr(A)[j+1]
    ib = getcolptr(B)[j]
    ib_nxt = getcolptr(B)[j+1]
    if ia < ia_nxt && ib < ib_nxt
      ra = rowvals(A)[ia]
      rb = rowvals(B)[ib]
      while true
        if ra < rb
          ia += oneunit(S1)
          ia < ia_nxt || break
          ra = rowvals(A)[ia]
        elseif ra > rb
          ib += oneunit(S2)
          ib < ib_nxt || break
          rb = rowvals(B)[ib]
        else # ra == rb
          r += dot(nonzeros(A)[ia],
              nonzeros(B)[ib])
          ia += oneunit(S1)
          ib += oneunit(S2)
          ia < ia_nxt && ib < ib_nxt || break
          ra = rowvals(A)[ia]
          rb = rowvals(B)[ib]
        end
      end
    end
  end
  return r
end
```

It is easy to rewrite it in a reversible style with NiLang without sacrificing much performance.

```julia
@i function dot(r::T, A::SparseMatrixCSC{T}, B::
     SparseMatrixCSC{T}) where {T}
    m ← size(A, 1)
    n ← size(A, 2)
    @invcheckoff branch_keeper ← zeros(Bool,2*m)
    @safe size(B) == (m,n) || throw(
     DimensionMismatch("matrices must have the
     same dimensions"))
    @invcheckoff @inbounds for j = 1:n
        ia1 ← A.colptr[j]
        ib1 ← B.colptr[j]
        ia2 ← A.colptr[j+1]
        ib2 ← B.colptr[j+1]
        ia ← ia1
        ib ← ib1
        @inbounds for i=1:ia2-ia1+ib2-ib1-1
            ra ← A.rowval[ia]
            rb ← B.rowval[ib]
            if (ra == rb, ~)
                r += A.nzval[ia]'*B.nzval[ib]
            end
            # b move -> true, a move -> false
            branch_keeper[i] ⊻= ia==ia2-1 ||
                ra > rb
            ra → A.rowval[ia]
            rb → B.rowval[ib]
            if (branch_keeper[i], ~)
                ib += identity(1)
            else
                ia += identity(1)
            end
        end

        ~@inbounds for i=1:ia2-ia1+ib2-ib1-1
            # b move -> true, a move -> false
            branch_keeper[i] ⊻= ia==ia2-1 ||
                A.rowval[ia] > B.rowval[ib]
            if (branch_keeper[i], ~)
                ib += identity(1)
            else
                ia += identity(1)
            end
        end
    end
    @invcheckoff branch_keeper → zeros(Bool, 2*m
     )
end
```

Here, all assignments are replaced with ← to indicate that the values of these variables must be returned at the end of this function scope. We put a "∼" symbol in the postcondition field of if statements to indicate this postcondition is a dummy one that takes the same value as the precondition, i.e. the condition is not changed inside the loop body. If the precondition is changed by the loop body, one can use a `branch_keeper` vector to cache branch decisions. The value of `branch_keeper` can be restored through uncomputing (the "∼" statement above). Finally, after checking the correctness of the program, one can turn off the reversibility checks by using the macro `@invcheckoff` macro to achieve better performance.

2.2 The first kind Bessel function

A Bessel function of the first kind of order ν can be computed via Taylor expansion

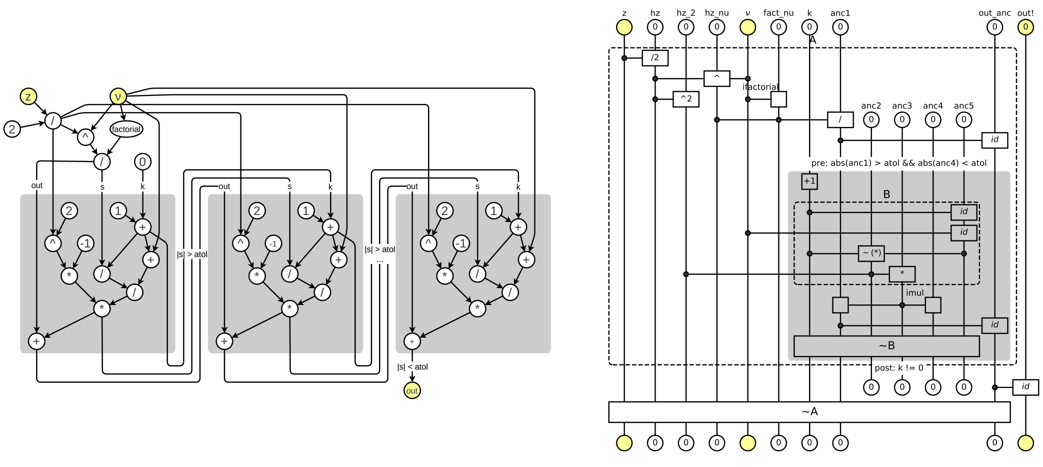
$$J_\nu(z) = \sum_{n=0}^{\infty} \frac{(z/2)^\nu}{\Gamma(\nu+1)\Gamma(k+\nu+1)}(-z^2/4)^n \tag{1}$$

where $\Gamma(n) = (n-1)!$ is the Gamma function. One can compute the accumulated item iteratively as $s_n = -\frac{z^2}{4}s_{n-1}$. The irreversible implementation is

```julia
function besselj(ν, z; atol=1e-8)
    k = 0
    s = (z/2)^ν / factorial(ν)
    out = s
    while abs(s) > atol
        k += 1
        s *= (-1) / k / (k+ν) * (z/2)^2
        out += s
    end
    out
end
```

This computational process could be diagrammatically represented as a computational graph as shown in Fig. 2 (a). The computational graph is a directed acyclic graph (DAG), where a node is a function and an edge is a data. An edge connects two nodes, one generates this data, and one consumes it. A computational graph is more likely a mathematical expression. It can not describe inplace functions and control flows conveniently because it does not have the notation for memory and loops.

With the logarithmic number system, we implement the reversible J_ν as follows.



(a) (b)

Figure 2: (a) The traditional computational graph for the irreversible implementation of the first kind Bessel function. A vertex (circle) is an operation, and a directed edge is a variable. The gray regions are the body of the unrolled while loop. (b) The memory oriented computational graph for the reversible implementation of the first kind Bessel function. Notations are explained in Fig. 4. The gray region is the body of a while loop. Its precondition and postcondition are positioned on the top and bottom, respectively.

```
@i function ibesselj(y!::T, ν, z::T; atol=1e-8)
    where T
  if z == 0
    if v == 0
      out! += 1
    end
  else
    @routine @invcheckoff begin
      k ← 0
      @ones ULogarithmic{T} lz halfz
          halfz_power_2 s
      @zeros T out_anc
      lz *= convert(z)
      halfz *= lz / 2
      halfz_power_2 *= halfz ^ 2
      # s *= (z/2)^ν/ factorial(ν)
      s *= halfz ^ ν
      for i=1:ν
        s /= i
      end
      out_anc += convert(s)
      while (s.log > -25, k!=0) # upto
          precision e^-25
        k += 1
        # s *= 1 / k / (k+ν) * (z/2)^2
        s *= halfz_power_2 / (k*(k+ν))
        if k%2 == 0
          out_anc += convert(s)
        else
          out_anc -= convert(s)
        end
      end
    end
    y! += out_anc
    ~@routine
  end
end
```

The above algorithm uses a constant number of ancillas, while the time overhead is also a constant. This reversible program can be diagrammatically represented as a memory oriented computational graph as shown in Fig. 2 (b). This diagram can be used to analyze variables uncomputing.

One can obtain gradients of this function by calling `Grad(ibesselj)`.

```
julia> out!, x = 0.0, 1.0
(0.0, 1.0)

julia> Grad(ibesselj)(Val(1), out!, 2, x)
(Val{1}(), GVar(0.0, 1.0), 2, GVar(1.0, 0.2102436))
```

Here, `Grad(ibesselj)` returns a callable instance of type `Grad{typeof(ibesselj)}`. The first parameters `Val(1)` specifies the position of loss in argument list. The Hessian can be obtained by feeding dual-numbers into this gradient function.

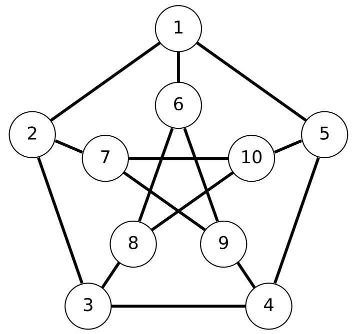


Figure 3: The Petersen graph has 10 vertices and 15 edges. We want to find a minimum embedding dimension for it.

```
julia> out!, x = 0.0, 1.0
(0.0, 1.0)

julia> Grad(ibesselj)(Val(1), out!, 2, x)
(Val{1}(), GVar(0.0, 1.0), 2, GVar(1.0, 0.2102436))

julia> using ForwardDiff: Dual

julia> _, hxout!, _, hxx = Grad(ibesselj)(Val(1),
           Dual(out!, zero(out!)), 2, Dual(x, one(x)));

julia> grad(hxx).partials[1]
0.13446683844358093
```

Here, the gradient field of `hxx` is defined as $\frac{\partial \text{out!}}{\partial x}$, which is a Dual number. It has a field `partials` that store the Hessian $\frac{\partial \text{out!}^2}{\partial x^2}$.

2.3 Solving a graph embedding problem

Graph embedding can be used to find a proper representation for an order parameter Takahashi and Sandvik (2020) in condensed matter physics. **??** considers a problem of finding the minimum Euclidean space dimension k that a Petersen graph can embed into, that the distances between pairs of connected vertices are l_1, and the distance between pairs of disconnected vertices are l_2, where $l_2 > l_1$. The Petersen graph is shown in Fig. 3. Let us denote the set of connected and disconnected vertex pairs as L_1 and L_2, respectively. This problem can be variationally solved with the following loss.

$$\begin{aligned}\mathcal{L} &= \mathrm{Var}(\mathrm{dist}(L_1)) + \mathrm{Var}(\mathrm{dist}(L_2)) \\ &+ \exp(\mathrm{relu}(\overline{\mathrm{dist}(L_1)} - \overline{\mathrm{dist}(L_2)} + 0.1))) - 1\end{aligned} \tag{2}$$

The first line is a summation of distance variances in two sets of vertex pairs, where $\mathrm{Var}(X)$ is the variance of samples in X. The second line is used to guarantee $l_2 > l_1$, where $\overline{X}$ means taking the average of samples in X. Its reversible implementation could be found in our benchmark repository.

We repeat the training for dimension k from 1 to 10. In each training, we fix two of the vertices and optimize the positions of the rest. Otherwise, the program will find the trivial solution with overlapped vertices. For $k < 5$, the loss is always much higher than 0, while for $k \geq 5$, we can get a loss close to machine precision with high probability. From the $k = 5$ solution, it is easy to see $l_2/l_1 = \sqrt{2}$. An Adam optimizer with a learning rate 0.01 Kingma and Ba requires ~ 2000 steps training. The trust region Newton's method converges much faster, which requires ~ 20 computations of Hessians to reach convergence. Although training time is comparable, the converged precision of the later is much better.

2.4 Gaussian mixture model and bundle adjustment

You will find the source code in github repositories https://github.com/JuliaReverse/NiGaussianMixture.jl and https://github.com/JuliaReverse/NiBundleAdjustment.jl.

3 Discussion and outlooks

In this main text, we show how to realize a reversible programming eDSL and implement an instruction level backward mode AD on top of it. It gives the user more flexibility to tradeoff memory and computing time comparing with traditional checkpointing. The Julia implementation NiLang gives the state-of-the-art performance and memory efficiency in obtaining first and second-order gradients in applications, including first type Bessel function, sparse matrix manipulations, solving graph embedding problem, Gaussian mixture model and bundle adjustment. It provides the possibility to differentiate GPU kernels. In the following, we discuss some practical issues about reversible programming, and several future directions to go.

3.1 Time Space Tradeoff

In history, there have been many discussions about time-space tradeoff on a reversible Turing machine (RTM). In the most straightforward g-segment tradeoff scheme Bennett (1989); Levine and Sherman (1990), an RTM model has either a space overhead that is proportional to computing time T or a computational overhead that sometimes can be exponential to the program size comparing with an irreversible counterpart. This result stops many people from taking reversible computing seriously as a high-performance computing scheme. In the following, we try to explain why the overhead of reversible computing is not as terrible as people thought.

First of all, the overhead of reversing a program is upper bounded by the checkpointing Chen *et al.* (2016) strategy used in many traditional machine learning package that memorizes inputs of primitives because checkpointing can be trivially implemented in reversible programming. Perumalla (2013) Reversible programming provides some alternatives to reduce the overhead. For example, accumulation is reversible, so that many BLAS functions can be implemented reversiblly without extra memory. Meanwhile, the memory allocation in some iterative algorithms can often be reduced by combining fixed point number and logarithmic numbers without sacrificing reversibility, as shown in the `ibesselj` example in Appendix 2.2. Clever compiling based on memory oriented computational graphs (Fig. 4 and Fig. 2 (b)) can also be used to help user tradeoff between time and space. The overhead of a reversible program mainly comes from the uncomputing of ancillas. It is possible to automatically uncompute ancillas by analyzing variable dependency instead of asking users to write `@routine` and `~@routine` pairs. In a hierarchical design, uncomputing can appear in every memory deallocation (or symbol table reduction). To quantify the overhead of uncomputing, we introduce the term uncomputing level as bellow.

***Definition* 1** (uncomputing level)**.** The log-ratio between the number of instructions of a reversible program and its irreversible counterpart.

To explain how it works, we introduce the memory oriented computational graph, as shown in Fig. 4. Notations are highly inspired by the quantum circuit representation. A vertical line is a variable and a horizontal line is a function. When a variable is used by a function, depending on whether its value is changed or not, we put a box or a dot at the line cross. It is different from the computational graph for being a hypergraph rather than a simple graph, because a variable can be used by multiple functions now. In panel (a). The subprogram in dashed box `X` is executed on space $x_{1:3}$ represents the computing stage. In the copying stage, the content in x_3 is read out to a pre-emptied memory x_4 through inplace add +=. Since this copy operation does not change contents of $x_{1:3}$, we can use the uncomputing operation ~`X` to undo all the changes to these registers. Now we computing the result x_4 without modifying the contents in $x_{1:3}$. If any of them is in a known state, it can be deallocated immediately. In panel (b), we can use the subprogram defined in (a) maked as Y to generate $x_{5:n}$ without modifying the contents of variables $x_{1:4}$. It is easy to see that although this uncompute-copy-uncompute design pattern can restore memories to known state, it has computational overhead. Both X and $\sim X$ are executed twice in the program (b), which is not

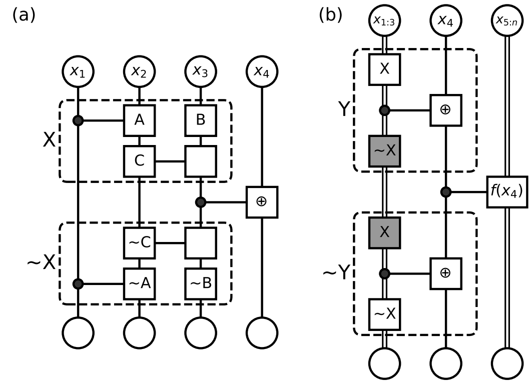


Figure 4: Two computational processes represented in memory oriented computational graph, where (a) is a subprogram in (b). In these graphs, a vertical single line represents one variable, a vertical double line represents multiple variables, and a parallel line represents a function. A dot at the cross represents a control parameter of a function and a box at the cross represents a mutable parameter of a function.

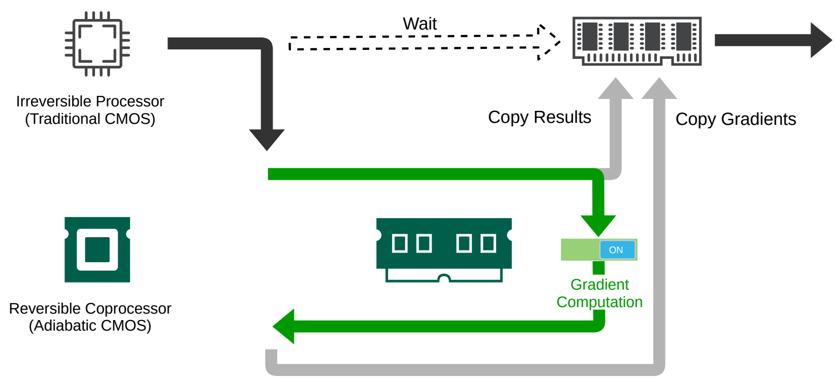


Figure 5: Energy efficient AI co-processor. Green arrows represents energy efficient operations on reversible devices.

necessary. We can cancel a pair of X and $\sim X$ (the gray boxes). By doing this, we are not allowed to deallocate the memory $x_{1:3}$ during computing $f(x_{5:n})$. This is the famous time-space tradeoff that playing the central role in reversible programming.

From the lowest instruction level, whenever we reduce the symbol table (or space), the computational cost doubles. The computational overhead grows exponentially as the uncomputating level increases, which can be seen from some of the benchmarks in the main text. In sparse matrix multiplication and dot product, we don't introduce uncomputing in the most time consuming part, so it is ~ 0. The space overhead is `2*m` to keep the branch decisions, which is even much smaller than the memory used to store row indices. in Gaussian mixture model, the most time consuming matrix-vector multiplication is doubled, so it is ~ 1. The extra memory usage is approximately 0.5% of the original program. In the first kind Bessel function and bundle adjustment program, the most time consuming parts are (nestedly) uncomputed twice, hence their uncomputing level is ~ 2. Such aggressive uncomputing makes zero memory allocation possible.

3.2 Differentiability as a Hardware Feature

So far, our eDSL is compiled to Julia language. It relies on Julia's multiple dispatch to differentiate a program, which requires users to write generic programs. A more liable AD should be a hardware or micro instruction level feature. In the future, we can expect NiLang being compiled to reversible instructions Vieri (1999) and executed on a reversible device. A reversible devices can play a role of differentiation engine as shown in the hetero-structural design in Fig. 5. It defines a reversible instruction set and has a switch that controls whether the instruction calls a normal instruction or an instruction that also updates gradients. When a program calls a reversible differentiable subroutine, the reversible co-processor first marches forward, compute the loss and copy the result to the main memory. Then the co-processor execute the program backward and uncall instructions, initialize and updating gradient fields at the same time. After reaching the starting point of the program, the gradients are transferred to the global memory. Running AD program on a reversible device can save energy. Theoretically, the reversible routines do not necessarily cost energy, the only energy bottleneck is copying gradient and outputs to the main memory.

3.3 The connection to Quantum programming

A Quantum device Nielsen and Chuang (2002) is a special reversible hardware that features quantum entanglement. The instruction set of classical reversible programming is a subset of quantum instruction set. However, building a universal quantum computer is difficult. Unlike a classical state, a quantum state can not be cloned. Meanwhile, it loses information by interacting with the environment. Classical reversible computing does not enjoy the quantum advantage, nor the quantum disadvantages of non-cloning and decoherence, but it is a model that we can try directly with our classical computer. It is technically smooth to have a reversible computing device to bridge the gap between classical devices and universal quantum computing devices. By introducing entanglement little by little, we can accelerate some elementary components in reversible computing. For example, quantum Fourier transformation provides an alternative to the reversible adders and multipliers by introducing the Hadamard and CPHASE quantum gates Ruiz-Perez and Garcia-Escartin (2017). From the programming languages's perspective, most quantum programming language preassumes the existence of a classical coprocessor to control quantum devices Svore *et al.* (2018). It is also interesting to know what is a native quantum control flow like, and does quantum entanglement provide speed up to automatic differentiation? We believe the reversible compiling technologies will open a door to study quantum compiling.

3.4 Gradient on ancilla problem

In this subsection, we introduce an easily overlooked problem in our reversible AD framework. An ancilla can sometimes carry a nonzero gradient when it is deallocated. As a result, the gradient program can be irreversible in the local scope. In NiLang, we drop the gradient field of ancillas instead of raising an error. In the following, we justify our decision by proving the following theorem.

***Theorem* 1.** *Deallocating an ancilla with constant value field and nonzero gradient field does not introduce incorrect gradients.*

Proof. Consider a reversible function $\mathbf{x}^i, b = f_i(\mathbf{x}^{i-1}, a)$, where a and b are the input and output values of an ancilla. Since both a, b are constants that are independent of input $\mathbf{x}^{i-1}$, we have

$$\frac{\partial b}{\partial \mathbf{x}^{i-1}} = \mathbf{0}. \tag{3}$$

Discarding gradients should not have any effect on the `value` fields of outputs. The key is to show `grad(b)` $\equiv \frac{\partial \mathbf{x}^L}{\partial b}$ does appear in the `grad` fields of the output. It can be seen from the back-propagation rule

$$\frac{\partial \mathbf{x}^L}{\partial \mathbf{x}^{i-1}} = \frac{\partial \mathbf{x}^L}{\partial \mathbf{x}^i}\frac{\partial \mathbf{x}^i}{\partial \mathbf{x}^{i-1}} + \frac{\partial \mathbf{x}^L}{\partial b}\frac{\partial b}{\partial \mathbf{x}^{i-1}}, \tag{4}$$

where the second term with $\frac{\partial \mathbf{x}^L}{\partial b}$ vanishes naturally. We emphasis here, the value part of discarded ancilla must be a constant. □

3.5 Shared read and write problem

One should be careful about shared read in reversible programming AD, because the shared read can introduce shared write in the adjoint program. Let's begin with the following expression.

```
y += x * y
```

Most people will agree that this statement is not reversible and should not be allowed because it changes input variables. We call it the *simultaneous read-and-write* issue. However, the following expression with two same inputs is a bit subtle.

```
y += x * x
```

It is reversible, but should not be allowed in an AD program because of the *shared write* issue. It can be seen directly from the expanded expression.

```
julia> macroexpand(Main, :(@instr y += x * x))
quote
    var"##253" = ((PlusEq)(*))(y, x, x)
    begin
        y = ((NiLangCore.wrap_tuple)(var"##253"))[1]
        x = ((NiLangCore.wrap_tuple)(var"##253"))[2]
        x = ((NiLangCore.wrap_tuple)(var"##253"))[3]
    end
end
```

In an AD program, the gradient field of `x` will be updated. The later assignment to x will overwrite the former one and introduce an incorrect gradient. One can get free of this issue by avoiding using same variable in a single instruction

```
anc ← zero(x)
anc += identity(x)
y += x * anc
anc -= identity(x)
```

or equivalently,

```
y += x ^ 2
```

Share variables in an instruction can be easily identified and avoided. However, it becomes tricky when one runs the program in a parallel way. For example, in CUDA programming, every thread may write to the same gradient field of a shared scalar. How to solve the shared write in CUDA programming is still an open problem, which limits the power of reversible programming AD on GPU.

3.6 Several future directions

We can use NiLang to solve some existing issues related to AD. We can use it to generate AD rules for existing machine learning packages like ReverseDiff, Zygote Innes *et al.* (2019), Knet Yuret (2016), and Flux Innes *et al.* (2018). Many backward rules for sparse arrays and linear algebra operations have not been defined yet in these packages. We can also use the flexible time-space tradeoff in reversible programming to overcome the memory wall problem in some applications. A successful, related example is the memory-efficient domain-specific AD engine in quantum simulator Yao Luo *et al.* (2019). This domain-specific AD engine is written in a reversible style and solved the memory bottleneck in variational quantum simulations. It also gives so far the best performance in differentiating quantum circuit parameters. Similarly, we can write memory-efficient normalizing flow Kobyzev *et al.* (2019) in a reversible style. Normalizing flow is a successful class of generative models in both computer vision Kingma and Dhariwal (2018) and quantum physics Dinh *et al.* (2016); Li and Wang (2018), where its building block bijector is reversible. We can use a similar idea to differentiate reversible integrators Hut *et al.* (1995); Laikov (2018). With reversible integrators, it should be possible to rewrite the control system in robotics Giftthaler *et al.* (2017) in a reversible style, where scalar is a first-class citizen rather than tensor. Writing a reversible control program should boost training performance. Reversibility is also a valuable resource for training.

To solve the above problems better, reversible programming should be improved from multiple perspectives. First, we need a better compiler for compiling reversible programs. To be specific, a compiler that admits mutability of data, and handle shared read and write better. Then, we need a reversible number system and instruction set to avoid rounding errors and support reversible control flows better. There are proposals of reversible floating point adders and multipliers, however these designs require allocating garbage bits in each operation Nachtigal *et al.* (2010, 2011); Nguyen and Meter (2013); Häner *et al.* (2018). In NiLang, one can simulate rigorous reversible arithmetic with the fixed-point number package FixedPointNumbers. However, a more efficient reversible design requires instruction-level support. Some other numbers systems are reversible under `*=` and `/=` rather than `+=` and `-=`, including LogarithmicNumbers Taylor *et al.* (1988) and TropicalNumbers. They are powerful tools to solve domain specific problems, for example, we have an upcoming work about differentiating over tropical numbers to solve the ground state configurations of a spinglass system efficiently. We also need `comefrom` like instruction as a partner of `goto` to specify the postconditions in our instruction set. Finally, although we introduced that the adiabatic CMOS as a better choice as the computing device in a spacecraft DeBenedictis *et al.* (2017). There are some challenges in the hardware side too, one can find a proper summary of these challenges in **??**.

Solutions to these issues requires the participation of people from multiple fields.

References

J. Takahashi and A. W. Sandvik, "Valence-bond solids, vestigial order, and emergent so(5) symmetry in a two-dimensional quantum magnet," (2020), arXiv:2001.10045 [cond-mat.str-el] .

D. P. Kingma and J. Ba, arXiv:1412.6980 .

C. H. Bennett, SIAM Journal on Computing **18**, 766 (1989).

R. Y. Levine and A. T. Sherman, SIAM Journal on Computing **19**, 673 (1990).

T. Chen, B. Xu, C. Zhang, and C. Guestrin, CoRR **abs/1604.06174** (2016), arXiv:1604.06174 .

K. S. Perumalla, *Introduction to reversible computing* (Chapman and Hall/CRC, 2013).

C. J. Vieri, *Reversible Computer Engineering and Architecture*, Ph.D. thesis, Cambridge, MA, USA (1999), aAI0800892.

M. A. Nielsen and I. Chuang, "Quantum computation and quantum information," (2002).

L. Ruiz-Perez and J. C. Garcia-Escartin, Quantum Information Processing **16** (2017), 10.1007/s11128-017-1603-1.

K. Svore, M. Roetteler, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, and A. Paz, Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018 (2018), 10.1145/3183895.3183901.

M. Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, and W. Tebbutt, CoRR **abs/1907.07587** (2019), arXiv:1907.07587 .

D. Yuret (2016).

M. Innes, E. Saba, K. Fischer, D. Gandhi, M. C. Rudilosso, N. M. Joy, T. Karmali, A. Pal, and V. Shah, "Fashionable modelling with flux," (2018), arXiv:1811.01457 [cs.PL] .

X.-Z. Luo, J.-G. Liu, P. Zhang, and L. Wang, "Yao.jl: Extensible, efficient framework for quantum algorithm design," (2019), arXiv:1912.10877 [quant-ph] .

I. Kobyzev, S. Prince, and M. A. Brubaker, "Normalizing flows: An introduction and review of current methods," (2019), arXiv:1908.09257 [stat.ML] .

D. P. Kingma and P. Dhariwal, in *Advances in Neural Information Processing Systems 31*, edited by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Curran Associates, Inc., 2018) pp. 10215–10224.

L. Dinh, J. Sohl-Dickstein, and S. Bengio, "Density estimation using real nvp," (2016), arXiv:1605.08803 [cs.LG] .

S.-H. Li and L. Wang, Physical Review Letters **121** (2018), 10.1103/physrevlett.121.260601.

P. Hut, J. Makino, and S. McMillan, The Astrophysical Journal **443**, L93 (1995).

D. N. Laikov, Theoretical Chemistry Accounts **137** (2018), 10.1007/s00214-018-2344-7.

M. Giftthaler, M. Neunert, M. Stäuble, M. Frigerio, C. Semini, and J. Buchli, Advanced Robotics **31**, 1225–1237 (2017).

M. Nachtigal, H. Thapliyal, and N. Ranganathan, in *10th IEEE International Conference on Nanotechnology* (2010) pp. 233–237.

M. Nachtigal, H. Thapliyal, and N. Ranganathan, in *2011 11th IEEE International Conference on Nanotechnology* (2011) pp. 451–456.

T. D. Nguyen and R. V. Meter, "A space-efficient design for reversible floating point adder in quantum computing," (2013), arXiv:1306.3760 [quant-ph] .

T. Häner, M. Soeken, M. Roetteler, and K. M. Svore, "Quantum circuits for floating-point arithmetic," (2018), arXiv:1807.02023 [quant-ph] .

F. J. Taylor, R. Gill, J. Joseph, and J. Radke, IEEE Transactions on Computers **37**, 190 (1988).

E. P. DeBenedictis, J. K. Mee, and M. P. Frank, Computer **50**, 76 (2017).