Differentiate Everything with a Reversible Domain-Specific Language

Jin-Guo Liu
Institute of Physics, Chinese Academy of Sciences,
Beijing 100190, China
cacate0129@iphy.ac.cn

Taine Zhao
Department of Computer Science, University of Tsukuba

Abstract

This paper considers implementing automatic differentiation (AD) in a reversible embedded domain-specific language. We start by reviewing the limitations of traditional AD frameworks. To solve the issues in these frameworks, we developed an open source reversible eDSL NiLang in Julia that can differentiate a general program while being compatible with Julia's ecosystem. It empowers users the flexibility to tradeoff time, space, and energy. With examples, we show one can use it to obtain gradients and Hessians for a wide class of functions in scientific programming and machine learning, including elementary mathematical functions, sparse matrix operaions and linear algebras (especially unitary matrices). Managable memory allocation makes it a good tool to differentiate GPU kernels. By benchmarking its performance in Bessel function, graph embedding problem, gaussian mixture model and bundle adjustment, we demonstrate that the AD implemneted in a reversible programming language can achieve state-of-the-art performance in both time and space. Finally, we will discuss the challenges that we face towards rigorous reversible programming, mainly from the instruction and hardware perspective.

1 Introduction

Computing the gradients of a numeric model $f : \mathbb{R}^m \to \mathbb{R}^n$ plays a crucial role in scientific computing. Consider a computing process

$$\begin{aligned}\mathbf{x}^1 &= f_1(\mathbf{x}^0)\\ \mathbf{x}^2 &= f_2(\mathbf{x}^1)\\ &\cdots\\ \mathbf{x}^L &= f_L(\mathbf{x}^{L-1})\end{aligned}$$

where $x^0 \in R^m$, $x^L \in R^n$, L is the depth of computing. The Jacobian of this program is a $n \times m$ matrix $J_{ij} \equiv \frac{\partial x_i^L}{\partial x_j^0}$, where x_j^0 and x_i^L are single elements of inputs and outputs. Computing part of the Jacobian automatically is what we called automatic differentiation (AD). It can be classified into three classes, the forward mode AD, the backward mode AD and the mixed mode AD. Hascoet and Pascual (2013) The forward mode AD computes the Jacobian matrix elements related to a single input using the chain rule $\frac{\partial \mathbf{x}^k}{\partial x_j^0} = \frac{\partial \mathbf{x}^k}{\partial \mathbf{x}^{k-1}} \frac{\partial \mathbf{x}^{k-1}}{\partial x_j^0}$ with j the column index, while a backward mode AD

Preprint. Under review.

computes Jacobian matrix elements related to a single output using $\frac{\partial \mathbf{x}_i^L}{\partial x^{k-1}} = \frac{\partial \mathbf{x}_i^L}{\partial \mathbf{x}^k}\frac{\partial \mathbf{x}^k}{\partial x^{k-1}}$ with i the row index. In variational applications where the loss function always outputs a scalar, the backward mode AD is preferred. However, implementing backward mode AD is harder than implementing its forward mode counterpart, because it requires propagating the gradients in the inverse direction of computing the loss. The backpropagation of gradients requires intermediate information of a program that includes

1. the computational process,
2. and variables used in computing gradients.

The computational process is often stored in a computational graph, which is a directed acyclic graph (DAG) that represents the relationship between data and functions. There are two basic techniques for the implementation of computational graph, which are operator overloading and source code transformation. Most popular AD implementations in the market are based on operator overloading. These packages provide a finite set of primitive functions with predefined backward rules. In Pytorch Paszke *et al.* (2017) and Flux Innes *et al.* (2018), every variable has a tracker field. When applying a predefined primitive function on a variable, the variable's tracker field keeps track of this function as well as data needed in later backpropagation. TensorFlow Abadi *et al.* (2015) uses a similar approach except it builds a static computational graph as a description of the program before actual computation happens. In research, people need new primitives frequently. Packages based on operator overloading can not cover all the diverse needs in different fields, hence it relies on users to code backward rules manually. For example, in physics, the requirements for AD are quite diverse.

1. We need to differentiate over sparse matrix operations that are important for Hamiltonian engineering Hao Xie and Wang, like solving dominant eigenvalues and eigenvectors Golub and Van Loan (2012).
2. We need to backpropagate singular value decomposition (SVD) function and QR decomposition in tensor network algorithms to study the phase transition problem Golub and Van Loan (2012); Liao *et al.* (2019).
3. We need to differentiate over a quantum simulation where each quantum gate is an inplace function that changes the quantum register directly Luo *et al.* (2019).

None of the above packages can meet all these requirements by using predefined primitives only. Scientists put lots of effort into deriving backward rules. In the backpropagation of dominant eigensolver Hao Xie and Wang, people managed to circumvent the sparse matrix issue by allowing users to provide the backward function for sparse matrices. The backward rules for SVD and eigensolvers have been formulated in recent years Seeger *et al.* (2017); Wan and Zhang (2019); Hubig (2019). One can obtain the gradients correctly for both real numbers and complex numbers Wan and Zhang (2019) in most cases, except when the spectrum is degenerate. In variational quantum simulator Yao Luo *et al.* (2019), authors implemented a builtin cache free AD engine by utilizing the reversible nature of quantum computing. They derive and implement the backward rule for each type of quantum gate.

Source code transformation based AD brings hope to free scientists from deriving backward rules. Tools like Tapenade Hascoet and Pascual (2013), ReverseDiff Rev and Zygote Innes (2018); Innes *et al.* (2019) generate the adjoint code statically while putting variable on a stack called the Wengert list. However, this approach has its problem too. In traditional AD applications, a program that might do billions of computations will get a Wengert list as well in the range of GBs. Frequent caching of data slows down the program significantly, and the memory will become a bottleneck as well. In both machine learning and scientific computation, memory management in AD is becoming a wall Luo *et al.* (2019) that limits the scale of many applications. In many deep learning models like recurrent neural network Lipton *et al.* (2015) and residual neural networks He *et al.* (2016), the depth can reach several thousand, where the memory is often the bottleneck of these programs. The memory wall problem is even more severe when one runs the application on GPU. The computational power of an Nvidia V100 GPU can reach 100 TFLOPS, which is comparable to a small cluster. However, its memory is only 32GB. Back propagating a general program using source code transformation makes the case worse because the memory consumption of the program typically scales as $O(T)$, where T is the runtime of the program. It is nearly impossible to automatically generate the backward rule

for SVD with the state-of-the-art performance. A better solution to memory management must be found to make source-to-source AD practical.

We tackle this problem by writing a program reversibly. Reversibility has been used in reducing the memory allocations in machine learning models such as recurrent neural networks MacKay *et al.* (2018), Hyperparameter learning Maclaurin *et al.* (2015) and residual neural networks Behrmann *et al.* (2018), where information buffer Maclaurin *et al.* (2015) and reversible activation functions Gomez *et al.* (2017); Jacobsen *et al.* (2018) are used to decrease the memory usage. Our approach makes reversibility a language feature so that it is a more general way of utilizing reversibility. We develop an embedded domain-specific language (eDSL) NiLang in Julia language Bezanson *et al.* (2012, 2017) that implements reversible programming. Perumalla (2013); Frank (2017a). This eDSL provides a macro to generate reversible functions that can be used by other programs. One can write reversible control flows, instructions, and memory managements inside this macro. We choose Julia as the host language for multiple purposes. Julia is a popular language for scientific programming. Its meta-programming and its package for pattern matching MLStyle MLS allow us to define an eDSL conveniently. Its type inference and just in time compiling can remove most overheads introduced in our eDSL, providing the state-of-the-art performance. Most importantly, its multiple-dispatch provides the polymorphism that will be used in our AD engine.

There have not been any reversible eDSL in Julia before, but there have been many prototypes of reversible languages like Janus Lutz (1986), R (not the popular one) Frank (1997), Erlang Lanese *et al.* (2018) and object-oriented ROOPL Haulund (2017). In the past, the primary motivation of studying reversible programming is to support reversible devices Frank and Knight Jr (1999) like adiabatic complementary metal–oxide–semiconductor (CMOS) Koller and Athas (1992), molecular mechanical computing system Merkle *et al.* (2018) and superconducting system Likharev (1977); Semenov *et al.* (2003). Reversible computing are more energy-efficient from the perspective of information and entropy, or by the Landauer's principle Landauer (1961). After decades of efforts, reversible computing devices are very close to providing productivity now. As an exmaple, adiabatic CMOS can be a better choice already in a spacecraft Hänninen *et al.* (2014); DeBenedictis *et al.* (2017), where energy is more valuable than device itself. Reversible programming is interesting to software engineers too, because it is a powerful tool to schedule asynchronious events Jefferson (1985) and debug a program bidirectionally Boothe (2000). However, the field of reversible computing faces the difficulty of not enough funding in recent decade Frank (2017a). As a result, not many people studying AD know the marvelous designs in reversible computing. People have not connected it with automatic differentiation seriously, even though they have many similarities. We aim to break the information barrier between the machine learning community and the reversible programming community in our work and provide yet another strong motivation to develop reversible programming.

In this paper, we first introduce the language design of NiLang in Sec. 2. In Sec. 3, we explain the back-propagation algorithm in this eDSL. In Sec. 4, we show several examples, including Bessel function, the dot product between sparse matrices, unitary matrix multiplication, and QR decomposition. We show how to generate first-order and second-order backward rules for these functions. We also show a practical application that solves the graph embedding problem. In Sec. 5, we benchmark the performance of NiLang with other AD Julia AD packages and Tapenade. In Sec. 6, we discuss several important issues, the time-space tradeoff, reversible instructions and hardware, and finally, an outlook to some open problems to be solved. In the appendix, we show the grammar of NiLang and other technical details.

2 Language design

2.1 Introductions to reversible language design

In a modern programming language, functions are pushed to a global stack for scheduling. The memory layout of a function consists of input arguments, a function frame with information like the return address and saved memory segments, local variables, and working stack. After the call, the function clears run-time information, only stores the return value. In reversible programming, this kind of design is no longer the best practice. One can not discard input variables and local

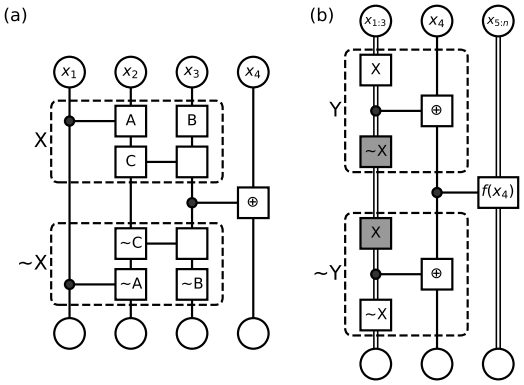


Figure 1: Two computational processes represented in memory oriented computational graph, where (a) is a subprogram in (b). In these graphs, a vertical single line represents one variable, a vertical double line represents multiple variables, and a parallel line represents a function. A dot at the cross represents a control parameter of a function and a box at the cross represents a mutable parameter of a function.

variables easily after a function call, since discarding information may ruin reversibility. For this reason, reversible functions are very different from irreversible ones from multiple perspectives.

2.1.1 Memory management

A distinct feature of reversible memory management is, the content of a variable must be known when it is deallocated. We denote the allocation of a zero emptied memory as $\mathtt{x} \leftarrow \mathtt{0}$, and the corresponding deallocation as $\mathtt{x} \rightarrow \mathtt{0}$. A variable x can be allocated and deallocated in a local scope, which is called an ancilla. It can also be pushed to a stack and used later with a pop statement. This stack is similar to a traditional stack, except it zero-clears the variable after pushing and presupposes that the variable being zero-cleared before popping.

Knowing the contents in the memory when deallocating is not easy. Hence Charles H. Bennett introduced the famous compute-copy-uncompute paradigm Bennett (1973). In order to show how reversible memory manage works, we introduce the memory oriented computational graph, as shown in Fig. 1. Notations are highly inspired by quantum circuit representations. A vertical line is a variable, and it can be used by multiple operations. Hence it is a hypergraph rather than a simple graph like DAG. When a variable is used by a function, depending on whether its value is changed, we put a box or a dot at the cross. Let us consider the example program shown in panel (a). The subprogram in dashed box $\mathtt{X}$ is executed on space $x_{1:3}$ to compute the desired result, which we call the computing stage. In the copying stage, the content in x_3 is read out to a pre-emptied memory x_4 through addition operation $\oplus$, and this is the piece of information that we care. Since this copy operation does not change contents of $x_{1:3}$, we can use the inverse operation $\sim\mathtt{X}$ to undo all the changes to these registers. If a variable in $x_{1:3}$ is initialized as a known value like 0, now it can be deallocated since its value is known again. If this subroutine of generating x_4 is used in another program as shown in Fig. 1 (b), x_4 can be uncomputed by reversing the whole subroutine in panel (a). The interesting fact is, both X and $\sim X$ are executed twice in this program, which seems to be unnecessary. We can, of course 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 $f(x_4)$, i.e., additional space is required. The tradeoff between space and time will be discussed in detail in Sec. 6.1.

2.1.2 Control flows

The reversible `if` statement is shown in Fig. 2 (a). It contains a precondition and a postcondition. The precondition decides which branch to enter in the forward execution, while the postcondition decides which branch to enter in the backward execution. After executing the specific branch, the program checks the consistency between precondition and postcondition to make sure they are consistent. The reversible `while` statement is shown in Fig. 2 (b). It also has both precondition and

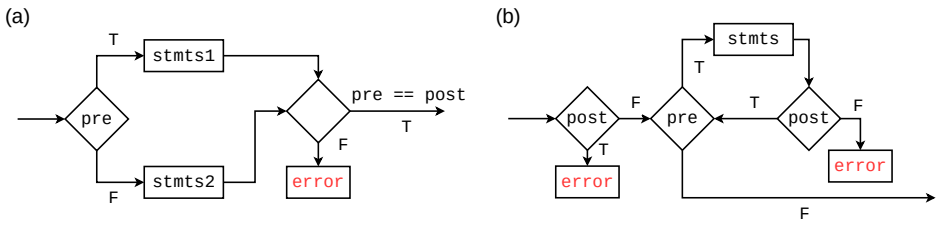


Figure 2: The flow chart for reversible (a) `if` statement and (b) `while` statement. "pre" and "post" represents precondition and postconditions respectively.

postcondition. Before executing the condition expressions, the program preassumes the postcondition is false. After each iteration, the program asserts the postcondition to be true. In the reverse pass, we exchange the precondition and postcondition. The reversible `for` statement is similar to irreversible ones except that after executing the loop, the program checks the values of these variables to make sure they are not changed. In the reverse pass, we exchange `start` and `stop` and inverse the sign of `step`.

2.1.3 Arithmetic instructions

Every arithmetic instruction has a unique inverse that can undo the changes. For logical operations, we have `y ⊻= f(args...)` self reversible. For other arithmetic operations, we regard `y += f(args...)` and `y -= f(args...)` as reversible to each other. Here `f` can be `identity`, `*`, `/` and `^` et. al. Besides the above two types of operations, `SWAP` operation that exchanges the contents in two memory spaces is also widely used in reversible computing systems. Here, it is worth noticing that `+=` and `-=` are not precisely reversible to each other because floating-point number operations have the rounding error. For applications sensitive to rounding errors, we should consider using other number systems, which will be discussed in Sec. 6.2.

2.2 NiLang

The main feature of NiLang is contained in a single macro `@i` that compiles a reversible function. The allowed statements in this eDSL are shown in Appendix A. We can use `macroexpand` to show the compiling a reversible function to the native Julia function.

```
julia> using NiLangCore, MacroTools

julia> ex = :(@i function f(x, y)
           SWAP(x, y)
       end)

julia> macroexpand(Main, ex) |> MacroTools.prettify
quote
    $(Expr(:meta, :doc))
    function $(Expr(:where, :(f(x, y))))
        koala = SWAP(x, y)
        x = (wrap_tuple(koala))[1]
        y = (wrap_tuple(koala))[2]
        (x, y)
    end
    if NiLangCore._typeof(f) != _typeof(~f)
        function $(Expr(:where, :((chimpanzee::_typeof(~f))(x, y))))
            loris = (~SWAP)(x, y)
            x = (wrap_tuple(loris))[1]
            y = (wrap_tuple(loris))[2]
            (x, y)
        end
    end
    if !(_hasmethod1(NiLangCore.isreversible, NiLangCore._typeof(f)))
        NiLangCore.isreversible(::NiLangCore._typeof(f)) = true
    end
end
```

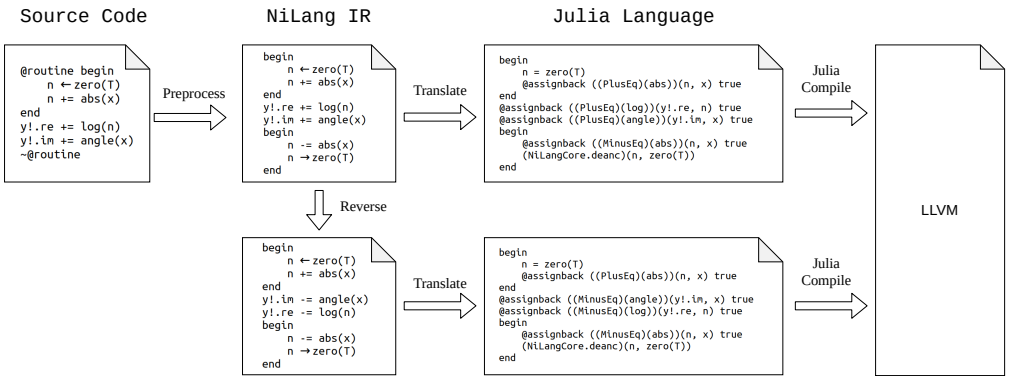


Figure 3: Compiling process of NiLang, the body of the complex valued log function as an example.

Here, the version of NiLang is v0.3.1. Macro `@i` generates three functions `f`, `~f` and `NiLangCore.isreversible`. `f` and `~f` are a pair of functions that are reversbile to each other. `~f` is an callable of type `Inv{typeof(f)}`, where the type parameter `typeof(f)` stands for th type of the function `f`. In the body of `f`, `NiLangCore.wrap_tuple` is used to unify output data types to tuples. The outputs of `SWAP` are assigned back to its input variables. At the end of this function, this macro attaches a return statement that returns all input variables. Finally, `NiLangCore.isreversible` marks `f` as reversible.

The compilation of a reversible function to native Julia functions is consisted of three stages: *preprocessing*, *reversing* and *translation*. Fig. 3 shows the compilation of the complex valued log function body, which is originally defined as follows.

Listing 1: Reversible implementation of the complex valued log function.

```
@i function ⊕(log)(y!::Complex{T}, x::Complex{T}) where T
    @routine begin
        n ← zero(T)
        n += abs(x)
    end
    y!.re += log(n)
    y!.im += angle(x)
    ~@routine
end
```

In the *preprocessing* stage, the compiler pre-processes human inputs to reversible NiLang IR. The preprocessor removes redundant grammars and expands shortcuts. It expands the "same as precondition" symbol ("~") in the postcondition field of an `if` statement by copying its precondition, adds missing ancilla deallocation statement ("←") to ensure the allocation and deallocation of an ancilla appear in pairs inside a local scope, and handles the computing-uncomputing macros `@routine` and `~@routine`. In the left most code box in Fig. 3, one uses `@routine <stmt>` statement to record a statement, and `~@routine` to insert the corresponding inverse statement for uncomputing. Here, one can input "←" and "→" in Julia by typing "\leftarrow[TAB KEY]" and "\rightarrow[TAB KEY]" respectively in a Julia editor or REPL.

In the *reversing* stage, based on this symmetric and reversible IR, the compiler generates reversed statements according to table Table 1. The reversible IR plays a central role in NiLang, from which one can see the allowed statements and how they are reversed.

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. We can expand the `@assignback` macro to see the compiled expression.

statement	inverse
`<f>(<args>...)`	`(~<f>)(<args>...)`
`<y> += <f>(<args>...)`	`<y> -= <f>(<args>...)`
`<y> .+= <f>.(<args>...)`	`<y> .-= <f>.(<args>...)`
`<y> ⊻= <f>(<args>...)`	`<y> ⊻= <f>(<args>...)`
`<y> .⊻= <f>.(<args>...)`	`<y> .⊻= <f>.(<args>...)`
`<a> ← <expr>`	`<a> → <expr>`
`(<T1> => <T2>)(<x>)`	`(<T2> => <T1>)(<x>)`

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

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

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

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

`@safe <expr>`	`@safe <expr>`

Table 1: The statements in NiLang IR, where elements in the left column and those in the right column are reversible to each other. "." is the symbol for the broadcasting magic in Julia, "~" is the symbol for reversing a statement or a function. `<pre>` stands for precondition, and `<post>` stands for postcondition "`begin <stmts> end`" is the statement for code block in Julia. It can be inverted by reversing the order as well as each element in it. We allow users to put an arbituary external statement inside a reversible context by putting a macro `@safe` in front of it. This statement is not reversible, but provides convenience. For example, one can use `@safe @show <var>` for debugging.

```
julia> macroexpand(Main, :(@assignback PlusEq(log)(y!.re, n)))
quote
    var"##277" = (PlusEq(log))(y!.re, n)
    begin
        y! = chfield(y!, Val{:re}(), ((NiLangCore.wrap_tuple)(var"##277"))[1])
        n = ((NiLangCore.wrap_tuple)(var"##277"))[2]
    end
end
```

Here, the function `chfield` returns a complex number with an updated `re` field. This updated value is then assigned back to `y!`. In other words, this macro simulates "inplace" operations on immutable types. Except fields, one can also define `chfield` on a function call and indexing. For example, `real(y!)` should also be inplace modifiable. We call an expression that directly modifiable in NiLang a *dataview*, it can be a variable itself, a field or an element of a dataview, or a bijective mapping of a dataview.

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.

One can also define a reversible constructor and destructor, we put this part in Appendix C.

3 Reversible automatic differentiation

3.1 First order gradient

Consider a computational process $\mathbf{x}^{i-1} = f_i^{-1}(\mathbf{x}^i)$ inside a reversed program, the Jacobians can be propagated in the reversed direction like

$$\begin{aligned} J^{\mathbf{x}^L}_{\mathbf{x}^{L'}} &= \delta_{\mathbf{x}^L,\mathbf{x}^{L'}}, \\ J^{\mathbf{x}^L}_{\mathbf{x}^{i-1}} &= J^{\mathbf{x}^L}_{\mathbf{x}^i} J^{\mathbf{x}^i}_{\mathbf{x}^{i-1}}, \end{aligned} \tag{1}$$

where $\mathbf{x}^L$ represents the outputs of the program. In backward mode AD, it is a scalar. $J^{\mathbf{x}^L}_{\mathbf{x}^i} \equiv \frac{\partial \mathbf{x}^L}{\partial \mathbf{x}^i}$ is the Jacobian to be propagated, and $J^{\mathbf{x}^i}_{\mathbf{x}^{i-1}}$ is the local Jacobian matrix. Einstein's notation Wikipedia contributors (2020a) is used here so that the duplicated index $\mathbf{x}^i$ in the second line is summed over. The algorithm to compute the backward mode AD can be summarized as follows.

Algorithm 1: Reversible Automatic Differentiation

Result: `grad`.(**x**)
let `iloss` be the index of the loss in **x**
x ← f(**x**)
for k = *1:length*(**x**) **do**
  x[k] ← `GVar`(**x**[k], $\delta_{k,\text{iloss}}$)
x ← f^{-1}(**x**)

We first compute the results with the forward pass $f(x)$. Then we wrap each output with a gradient field $\delta_{k,\text{iloss}}$, which is the Dirac delta notation. The gradient field is initialized to 1 if the variable is the loss else 0. The new number type with a gradient field is called `GVar`. If the `GVar` constructor meets an array, it will be broadcasted to each element of this array. Then we feed these `GVar` instances into the backward pass f^{-1}. Finally, the gradients can be accessed with the `grad` dataview of output variables. Here we emphasis that, in the backward pass, since the basic element types are changed, different instructions are called. The new instructions update gradient fields for variables during computing. This is the multiple-dispatch in Julia that a function can be dynamically dispatched based on the run time type of more than one of its arguments. Similar approach has been used in the forward mode AD package ForwardDiff Revels *et al.* (2016). As an example, to bind the backward rules for instructions $\oplus(*)$ (or `PlusEq(*)`) and $\ominus(*)$ (or `MinusEq(*)`). One can overload **either** of them as follows.

```
@i function ⊖(*)(out!::GVar, x::GVar, y::GVar)
    value(out!) -= value(x) * value(y)
    grad(x) += grad(out!) * value(y)
    grad(y) += value(x) * grad(out!)
end
```

Here, the first line in the function body does normal computing for the `value` dataview. The second and thrid lines update the gradient fields of `x` and `y`, where update rule corresponds to the backward rule of $\oplus(*)$. The update rule defined on $\oplus(*)$ is automatically generated by macro `@i`, which reflects the fact that taking inverse and computing gradients commute McInerney (2015). One can check the correctness of this definition as follows.

```
julia> using NiLang, NiLang.AD

julia> a, b, y = GVar(0.5), GVar(0.6), GVar(0.9)
(GVar(0.5, 0.0), GVar(0.6, 0.0), GVar(0.9, 0.0))

julia> @instr grad(y) += identity(1.0)

julia> @instr y += a * b
GVar(0.6, -0.5)

julia> a, b, y
(GVar(0.5, -0.6), GVar(0.6, -0.5), GVar(1.2, 1.0))

julia> @instr y -= a * b
GVar(0.6, 0.0)

julia> a, b, y
(GVar(0.5, 0.0), GVar(0.6, 0.0), GVar(0.899999, 1.0))
```

Here, since $J(\ominus(*)) = J(\oplus(*))^{-1}$, consecutively applying them will restore the gradient fields of all variables. More local Jacobians and Hessians for basic instructions used in this section could be found in Appendix C.1.

3.2 Second-order gradient

Combining the adjoint program in NiLang with dual-numbers is a simple yet efficient way to obtain Hessians. By wrapping the elementary type with `Dual` defined in package ForwardDiff Revels *et al.* (2016) and throwing it into the gradient program defined in NiLang, one obtains one row/column of the Hessian matrix straightforward. We will show an example of using forward differentiating in Newton's trust region optimization in Sec. 4.5.

3.3 Differentiating complex numbers

To differentiate complex numbers, we re-implemented complex instructions reversiblly. For example, with the definition of complex valued log function in Listing. 1, the complex valued log can be differentiated with no extra effort.

4 Examples

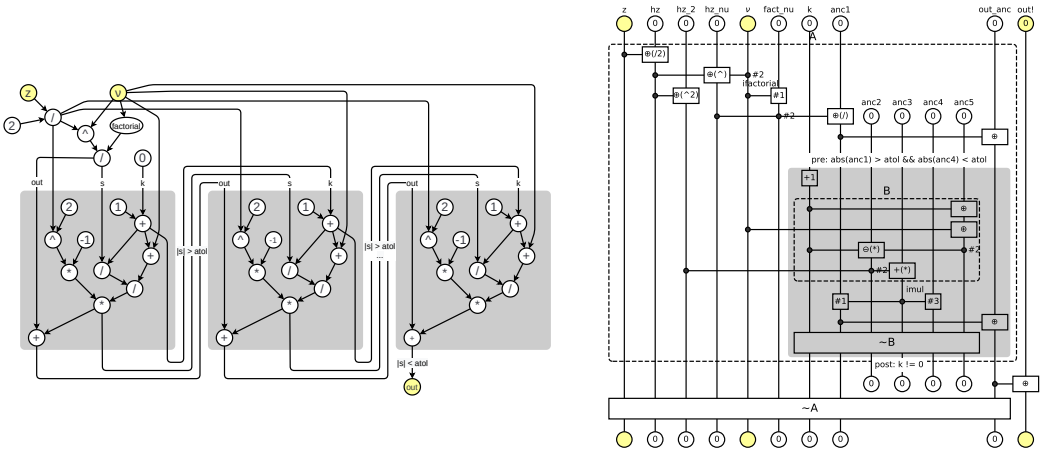
In this section, we introduce several examples. We will discuss the first example, the first kind Bessel function, in detail. We compare the difference between the irreversible and reversible implementations of this function, as well as the difference between regular computational graph and memory oriented computational graph. Then we show how to obtain first and second-order gradients automatically in the reversible AD framework. We benchmark different source-to-source AD implementations of the Bessel function. Then we show how to differentiate sparse matrix operations, unitary matrix multiplication, and QR decomposition. Finally, we show how to solve the graph embedding problem variationally.

4.1 The first kind Bessel function

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

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

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



(a) (b)

Figure 4: (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. 1. The gray region is the body of a while loop. Its precondition and postcondition are positioned on the top and bottom, respectively.

```
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 DAG as shown in Fig. 4 (a). In this diagram, the data is represented as an edge. It connects at most two nodes. One generates this data, and one consumes it. A computational graph is more likely a mathematical expression, and it can not describe inplace functions or control flows conveniently because it does not have the notation for memory and loops.

In the following, we introduce the reversible implementation and the memory oriented computational graph. The above Bessel function contains a loop with irreversible "*=" operation inside. Intuitively, consequtive multiplication requires an increasing size of tape to cache the intermediate state s_n, since one can not release state s_{n-1} directly after computing s_n Perumalla (2013). To reduce the memory allocation without increasing the time complexity of the program, we introduce the following reversible approximate multiplier.

```
1 @i @inline function imul(out!, x, anc!)
2     anc! += out! * x
3     out! -= anc! / x
4     SWAP(out!, anc!)
5 end
```

Here, instruction `SWAP` exchanges values of the two variables, and `anc!` ≈ 0 is a *dirty ancilla*. Line 2 computes the result and accumulates it to the dirty ancilla, and we get an approximately correct output in `anc!`. Line 3 uncomputes `out!` approximately by using the information stored in `anc!`,

leaving a dirty zero state in register `out!`. Line 4 swaps the contents in `out!` and `anc!`. Finally, we have an approximately correct output and a dirtier ancilla. The "approximate uncomputing" trick can be extensively used in practice. It mitigates the artificial irreversibility brought by the number system that we have adopted at the cost of output precision. The reason why this trick works here lies in the fact that from the mathematics perspective the state in nth step $\{s_n, z\}$ contains the same amount of information as its previous state $\{s_{n-1}, z\}$ except for some particular points, and it is highly possible to find an equation to uncompute the previous state from the current state. With this approximate multiplier, we implement J_ν as follows.

```
using NiLang, NiLang.AD

@i function ibesselj(out!, ν, z; atol=1e-8)
    k ← 0
    fact_nu ← zero(ν)
    halfz ← zero(z)
    halfz_power_nu ← zero(z)
    halfz_power_2 ← zero(z)
    out_anc ← zero(z)
    anc1 ← zero(z)
    anc2 ← zero(z)
    anc3 ← zero(z)
    anc4 ← zero(z)
    anc5 ← zero(z)

    @routine begin
        halfz += z / 2
        halfz_power_nu += halfz ^ ν
        halfz_power_2 += halfz ^ 2
        ifactorial(fact_nu, ν)
        anc1 += halfz_power_nu/fact_nu
        out_anc += identity(anc1)
        while (abs(unwrap(anc1))>atol && abs(
      unwrap(anc4)) < atol, k!=0)
            k += identity(1)
            @routine begin
                anc5 += identity(k)
                anc5 += identity(ν)
                anc2 -= k * anc5
                anc3 += halfz_power_2 / anc2
            end
            imul(anc1, anc3, anc4)
            out_anc += identity(anc1)
            ~@routine
        end
    end
    out! += identity(out_anc)
    ~@routine
end
```

Here, the definition of `ifactorial` could be found in the appendix. Comparing with it irreversible counterpart, the number of additional ancillas is a constant, while the time overhead factor is also a constant. Ancilla `anc4` plays the role of *dirty ancilla* in multiplication, and it is uncomputed rigorously in the uncomputing stage marked by `~@routine`.

This reversible program can be diagrammatically represented as a memory oriented computational graph as shown in Fig. 4 (b). In this graph, a variable is a vertical line, while a function is a parallel line. The critical difference comparing with the traditional computational graph is that it adopts a variable oriented view. A variable can be accessed by multiple functions. Hence it represents a hypergraph rather than a simple graph. If a function uses a variable but does not change the contents in it, we call this variable a control parameter of this function and put a dot at the cross. Otherwise, if the content is changed, we put a square. This diagram can be used to analyse uncomputable variables. In this example routine "B" uses `hz_2`, ν and `k` as control parameters, and changes the contents in `anc2`, `anc3` and `anc5`. while the following operation `imul` does not change these variables. Hence we can apply the inverse routine `~B` to safely restore contents in `anc2`, `anc3` and `anc5`, and this is what people called compute-copy-uncompute paradigm.

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.

```
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 derivative for `x`. It corresponds to the Hessian $\frac{\partial \text{out!}^2}{\partial x^2}$ that we need.

4.2 Sparse Matrices

Source to source automatic differentiation is useful in differentiating sparse matrices. It is a well-known problem that sparse matrix operations can not benefit directly from generic backward rules for dense matrix because general rules do not keep the sparse structure. In the following, we will show that reversible AD can differentiate the Frobenius dot product between two sparse matrices with the state-of-the-art performance. Here, the Frobenius dot product is defined as `trace(A'B)`. This following reversible implementation is adapted from the irreversible implementation in Julia package `SparseArrays`.

```
using SparseArrays

@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
```

With simple adaptation, the code becomes reversible. Here, the key point is using a `branch_keeper` vector to cache branch decisions.

4.3 Unitary Matrices

A unitary matrix features uniform eigenvalues and reversibility. It is widely used as an approach to ease the gradient exploding and vanishing problem Arjovsky *et al.* (2015); Wisdom *et al.* (2016); Jing *et al.* (2016) and the memory wall problem Luo *et al.* (2019). One of the simplest ways to parametrize a unitary matrix is representing a unitary matrix as a product of two-level unitary operations Jing *et al.* (2016). A real unitary matrix of size N can be parametrized compactly by $N(N-1)/2$ rotation operations LI *et al.* (2013)

$$\text{ROT}(a!, b!, \theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} a! \\ b! \end{pmatrix}, \tag{3}$$

where θ is the rotation angle, `a!` and `b!` are target registers.

```
using NiLang, NiLang.AD

@i function umm!(x!, θ)
    @safe @assert length(θ) ==
            length(x!)*(length(x!)-1)/2
    k ← 0
    for j=1:length(x!)
        for i=length(x!)-1:-1:j
            k += identity(1)
            ROT(x![i], x![i+1], θ[k])
        end
    end

    k → length(θ)
end
```

Here, the ancilla `k` is deallocated manually by specifying its value, because we know the loop size is $N(N-1)/2$. We define the test functions in order to check gradients.

```
julia> @i function test!(out!, x!::Vector, θ::
     Vector)
           umm!(x!, θ)
           isum(out!, x!)
       end

julia> out, x, θ = 0.0, randn(4), randn(6);

julia> @instr Grad(test!)(Val(1), out, x, θ)

julia> x
4-element Array{GVar{Float64,Float64},1}:
 GVar(1.220182125326287, 0.14540743042341095)
 GVar(2.1288634811475937, -1.3749962375499805)
 GVar(1.2696579252569677, 1.42868739498625)
 GVar(0.1083891125379283, 0.2170123344615735)

julia> @instr (~test!')(Val(1), out, x, θ)

julia> x
4-element Array{Float64,1}:
 1.220182125326287
 2.1288634811475933
 1.2696579252569677
 0.10838911253792821
```

In the above testing code, `Grad(test)` attaches a gradient field to each element of `x`. `~Grad(test)` is the inverse program that erase the gradient fields. Notably, this reversible implementation costs zero memory allocation, although it changes the target variables inplace.

4.4 QR decomposition

Let us consider a naive implementation of QR decomposition from scratch. We admit this implementation is just a proof of principle which does not consider reorthogonalization and other practical issues.

```
using NiLang, NiLang.AD

@i function qr(Q, R, A::Matrix{T}) where T
    anc_norm ← zero(T)
    anc_dot ← zeros(T, size(A,2))
    ri ← zeros(T, size(A,1))
    for col = 1:size(A, 1)
        ri .+= identity.(A[:,col])
        for precol = 1:col-1
            dot(anc_dot[precol], Q[:,precol], ri
     )
            R[precol,col] +=
                identity(anc_dot[precol])
            for row = 1:size(Q,1)
                ri[row] -=
                    anc_dot[precol] * Q[row,
     precol]
            end
        end
        norm2(anc_norm, ri)
        R[col, col] += anc_norm^0.5
        for row = 1:size(Q,1)
            Q[row,col] += ri[row] / R[col, col]
        end

        ~begin
            ri .+= identity.(A[:,col])
            for precol = 1:col-1
                dot(anc_dot[precol], Q[:,precol]
    , ri)
                for row = 1:size(Q,1)
                    ri[row] -= anc_dot[precol] *
                        Q[row, precol]
                end
            end
            norm2(anc_norm, ri)
        end
    end
end
```

Here, in order to avoid frequent uncomputing, we allocate ancillas `ri` and `anc_dot` as vectors. The expression in ~ is used to uncompute `ri`, `anc_dot` and `anc_norm`. `dot` and `norm2` are reversible functions to compute dot product and vector norm. One can quickly check the correctness of the gradient function

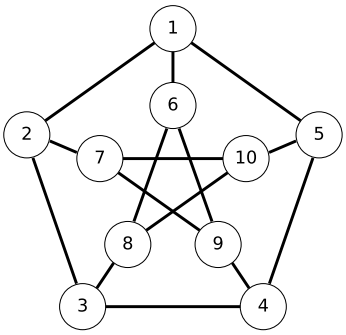


Figure 5: The Petersen graph has 10 vertices and 15 edges. We want to find a minimum embedding dimension for it.

```
julia> A  = randn(4,4);

julia> q, r = zero(A), zero(A);

julia> @i function test1(out, q, r, A)
           qr(q, r, A)
           isum(out, q)
       end

julia> check_grad(test1, (0.0, q, r, A); iloss=1)
true
```

Here, the loss function `test1` is defined as the sum of the output unitary matrix `q`. The `check_grad` function is a gradient checker function defined in module NiLang.AD.

4.5 Solving a graph embedding problem

Graph embedding can be used to find representation for an order parameter Takahashi and Sandvik (2020) in condensed matter physics. Ref. Takahashi and Sandvik (2020) considers a problem of finding the minimum Euclidean space dimension k that a Petersen graph can fit into, with extra requirements that the distance between a pair of connected vertices has the same value l_1, and the distance between a pair of disconnected vertices has the same value l_2 and $l_2 > l_1$. The Petersen graph is ten vertices graph, as shown in Fig. 5. Let us denote the set of connected and disconnected vertex pairs as L_1 and L_2, respectively. This problem can be variationally solved by differential programming by designing the subsequent loss.

$$\begin{aligned}\mathcal{L} =& \text{Var}(\text{dist}(L_1)) + \text{Var}(\text{dist}(L_2)) \\ &+ \exp(\text{relu}(\overline{\text{dist}(L_1)} - \overline{\text{dist}(L_2)} + 0.1))) - 1\end{aligned} \tag{4}$$

The first line is a summation of distance variances in two sets of vertex pairs, where Var$X$ means taking 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 each dimension k from 1 to 10 and search for possible solutions by variationally optimizing the positions of vertices. In each training, we fix two of the vertices and train the rest. Otherwise, the program will find the trivial solution with overlapped vertices. For $k = 5$, we can get a loss close to machine precision with high probability, while for $k < 5$, the loss is always much higher than 0. From the solution, it is easy to see $l_2/l_1 = \sqrt{2}$ is the solution. For $k = 5$, an Adam optimizer with a learning rate 0.01 Kingma and Ba requires $\sim$ 2000 steps training. The trust region Newton's method converges much faster, which requires $\sim$ 20 computations of Hessians to reach convergence. Although training time is comparable, the converged precision of the later is much better.

5 Benchmark

In the following benchmarks the CPU device is Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz, and the GPU device is Nvidia Titan V. For NiLang benchmarks, we have turned off the reversibility check off to achieve better performance.

5.1 Bessel Function

We differentiate the first type Bessel function in Sec. 4.1 and show the benchmarks in Table 2.

	$T_{\min}$/ns	Space/KB
Julia-O	17	0
NiLang-O	53	0
Tapenade-O	32	0
ForwardDiff-G	39	0
NiLang-G	231	0
NiLang-G (CUDA)	1.4	0
ReverseDiff-G	7198	7.3
Zygote-G	22561	13.47
Tapenade-G (Forward)	30	0
Tapenade-G (Backward)	111	> 0

Table 2: Time and space used for computing objective (O) and gradient (G) of the first kind Bessel function $J_2(1.0)$.

In the table, Julia is the CPU time used for running the irreversible forward program. It is the baseline for benchmarking. NiLang (call/uncall) is the time of reversible call or uncall. Both of them are ~ 3 times slower than its irreversible counterpart. Here, we have removed the reversibility check to avoid overheads. One can always turn off this check after debugging. Since Bessel function has only one input argument, forward mode AD tools are faster than reverse mode AD, both source-to-source framework ForwardDiff and operator overloading framework Tapenade have the a comparable computing time with the pure function call. NiLang.AD is the reverse mode AD submodule in NiLang, and it takes 13.6 times the native Julia program, and is also 2 times slower than Tapenade. However, the key point is, there is no extra memory allocation like stack operations in the whole computation. The controllable memory allocation of NiLang makes it compatible with CUDA program. In other backward mode AD like Zygote, ReverseDiff and Tapenade, the memory allocation in heap is nonzero due to the checkpointing and possible failure of type inference of Julia language. NiLang is friendly to type inference because it is closure free.

5.2 Graph embedding problem

Since the ForwardDiff itself provides the Hessian for users, it is interesting to benchmark how much performance we can get by forward differentiating an adjoint program comparing with forward differentiating a forward AD program. In the following benchmark, we show the benchmark for the graph embedding problem in Sec. 4.5.

In this application, the number of input parameters scales as $10 \times k$, where k is the embedding dimension of the graph. In Table 3, we show the the performance of different implementations by varying the dimension k. As the baseline, (a) shows the time for computing the 0th-order gradient, or the function call. We have reversible and irreversible implementations, where the reversible program is slower than the irreversible native Julia program by a factor of ~ 2. (b) shows the time for computing the first-order gradients. The reversible program shows the advantage of obtaining gradients when the dimension $k \geq 3$. The larger the number of inputs, the more advantage it shows due to the overhead proportional to input size in forward mode AD. The same reason applies to computing Hessians. The mixed-mode AD gives better performance when $k \geq 3$ comparing with

	2	4	6	8	10
Julia-O	4.477e-06	4.729e-06	4.959e-06	5.196e-06	5.567e-06
NiLang-O	7.173e-06	7.783e-06	8.558e-06	9.212e-06	1.002e-05
NiLang-U	7.453e-06	7.839e-06	8.464e-06	9.298e-06	1.054e-05
NiLang-G	1.509e-05	1.690e-05	1.872e-05	2.076e-05	2.266e-05
ForwardDiff-G	1.518e-05	4.053e-05	6.732e-05	1.184e-04	1.701e-04
ReverseDiff-G	1.384e-04	1.928e-04	2.392e-04	2.893e-04	3.556e-04
Zygote-G	5.315e-04	5.570e-04	5.811e-04	6.096e-04	6.396e-04
(NiLang+F)-H	4.528e-04	1.025e-03	1.740e-03	2.577e-03	3.558e-03
ForwardDiff-H	2.378e-04	2.380e-03	6.903e-03	1.967e-02	3.978e-02
(ReverseDiff+F)-H	1.966e-03	6.058e-03	1.225e-02	2.035e-02	3.140e-02

Table 3: Absolute runtimes in seconds for computing the objectives (O), uncall objective (U), gradients (G) and Hessians (H) of the graph embedding program.

pure forward mode AD. Comparing with other backward mode AD packages ReverseDiff and Zygote, NiLang is approximately one order more efficient for the same reason we discussed in Sec. 4.1.

5.3 Sparse matrices

We benchmarked the call, uncall and backward time used for sparse matrix dot product and matrix multiplication. Here, we estimate the time for back propagating gradients rather than including both forward and backward. This is prefered because `mul!` usually plays a role of an intermediate step of computing, which does not output a scalar as loss.

	dot	mul! (complex valued)
Julia-O	3.493e-04	8.005e-05
NiLang-O	4.675e-04	9.332e-05
NiLang-B	5.821e-04	2.214e-04

Table 4: Absolute runtimes in seconds for computing the objectives (O) and the backward pass (B) of sparse matrix operations. The matrix size is 1000×1000, and the element density is 0.05. The total time used in computing gradient can be estimated as a sum of times in row "O" (reversible or not) and row "B".

The time used for computing backward pass is approximately 1.5-3 times the Julia's native forward pass. This is because the instruction length of differentiating basic arithemetic instructions is longer than pure computing by a factor of 2 or more.

5.4 Gaussian mixture model and bundle adjustment

We reproduced the benchmarks for Gaussian mixture model (GMM) and bundle adjustment (BA) in Srajer *et al.* (2018) by re-writting the programs in a reversible style. We show the results in Table 5 and Table 6. Notice in these benchmarks, we rewrite the ForwardDiff program for a fair benchmark, this explains the difference between our results and the original benchmark. The Tapenade data is obtained by executing the docker file provided by the original benchmark, which is the baseline of our benchmark and the original benchmark.

In this case, NiLang shows slight advantage over forward mode AD because the bottleneck of computing this large sparse Jacobian is computing the Jacobian of a elementary function with 15 input arguments and 2 output arguments, where input space is larger than output space.

# parameters	3.00e+1	3.30e+2	1.20e+3	3.30e+3	1.07e+4	2.15e+4	5.36e+4	4.29e+5
Julia-O	9.189e-03	1.193e-02	2.494e-01	8.618e-02	3.523e-02	7.641e-02	2.254e-01	3.404e+00
NiLang-O	1.657e-02	5.009e-02	4.902e-01	4.625e-01	3.036e-01	6.095e-01	1.594e+00	1.529e+01
Tapende-O	1.484e-03	3.747e-03	4.836e-02	3.578e-02	5.314e-02	1.069e-01	2.583e-01	2.200e+00
ForwardDiff-G	3.360e-02	1.240e+00	3.984e+01	1.429e+02	-	-	-	-
NiLang-G	3.510e-02	1.136e-01	1.064e+00	1.066e+00	1.700e+00	3.328e+00	8.643e+00	7.354e+01
Tapenade-G	5.484e-03	1.434e-02	2.205e-01	1.497e-01	4.396e-01	9.588e-01	2.586e+00	2.442e+01

Table 5: Absolute runtimes in seconds for computing the objective (O) and gradients (G) of GMM with 10k data points.

# measurements	3.18e+4	2.04e+5	2.87e+5	5.64e+5	1.09e+6	4.75e+6	9.13e+6
Julia-O	2.020e-03	1.292e-02	1.812e-02	3.563e-02	6.904e-02	3.447e-01	6.671e-01
NiLang-O	2.708e-03	1.757e-02	2.438e-02	4.877e-02	9.536e-02	4.170e-01	8.020e-01
Tapenade-O	1.632e-03	1.056e-02	1.540e-02	2.927e-02	5.687e-02	2.481e-01	4.780e-01
ForwardDiff-J	6.579e-02	5.342e-01	7.369e-01	1.469e+00	2.878e+00	1.294e+01	2.648e+01
NiLang-J	1.651e-02	1.182e-01	1.668e-01	3.273e-01	6.375e-01	2.785e+00	5.535e+00
NiLang-J (GPU)	1.354e-04	4.329e-04	5.997e-04	1.735e-03	2.861e-03	1.021e-02	2.179e-02
Tapenade-J	1.940e-02	1.255e-01	1.769e-01	3.489e-01	6.720e-01	2.935e+00	6.027e+00

Table 6: Absolute runtimes in seconds for computing the objective (O) and Jacobians (J) in bundle adjustment.

6 Discussion and outlook

In this paper, we show how to realize a reversible programming eDSL and how to implement source-to-source 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, linear algebra functions and, a practical one, the application graph embedding problem.

In the following, we discuss some practical issues about reversible programming, and several future directions to go.

6.1 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 convince the readers that the overhead of reversible computing is not as terrible as people thought.

The overhead of reversing a program is bounded by the checkpointing Chen *et al.* (2016) strategy used in a traditional machine learning package that memorizes inputs of primitives because similar strategy can also be used in reversible programming. Perumalla (2013) Reversible programming provides more alternatives to reduce the overhead. For example, accumulation is reversible, and it does not require checkpointing. The checkpointing in many iterative algorithms can often be avoided with the "arithmetic uncomputing" trick without sacrificing reversibility, as shown in the `ibesselj` example in Sec. 4.1.

As shown in Fig. 1, clever compiling based on memory oriented computational graphs can also be used to help user tradeoff between time and space. Often, when we define a new reversible function, we allocate some ancillas at the beginning of the function and deallocate them through uncomputing at the end. The overhead comes from the uncomputing. In the worst case, the time used for uncomputing can be the same as the normal call. In a hierarchical design, uncomputing can appear in every layer of the abstraction. To quantify the overhead of uncomputing, we introduce the term program granularity as bellow.

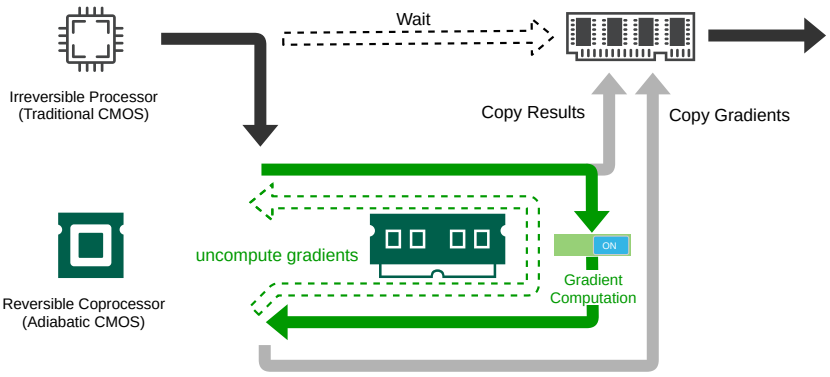


Figure 6: Energy efficient AI co-processor. Green arrows represents energy efficient operations on reversible devices. Dashed lines does not occupy the CPU time.

***Definition* 1** (program granularity)**.** The log-ratio between the execution time of a reversible program and its irreversible counterpart.

The computing time increases exponentially as the granularity increases. A cleverer compilation of a program can reduce the granularity by merging the uncomputing statements to avoid repeated efforts.

At last, making reversible programming an eDSL rather than an independent language allows flexible choices between reversibility and computational overhead. For example, to deallocate the memory that stores gradients in a reversible language, one has to uncompute the whole process of obtaining them. As an eDSL, one has an alternative to deallocate the memory irreversibly outside the scope of a reversible program, i.e., trade energy with time.

6.2 Differentiability as a Hardware Feature

So far, our eDSL is compiled to Julia. In the future, it can be compiled to reversible instructions Vieri (1999) and executed on a reversible device. We propose a reversible-irreversible hetero-structural hardware design for differential programming, where a reversile device plays the role of energy efficient gradient provider. A reversible device defines a reversible instruction set. It has a switch that controls whether the instruction calls a normal instruction or an instruction that also updates gradients. As show in Fig. 6, when a program calls a reversible routine, the reversible program is compiled to a reversible instruction set. The reversible co-processor reads the instruction set forward with gradient switch off, and copy the result to global memory. Then it reads the instruction set backward and uncall each instruction with gradient switch on. Since the gradient switch is on, it opens a new space for gradient updating. After reaching the starting of the program, the gradient is computed, we copy the desired parts to global memory. With the output and gradient information, the main processor can keep going, while the reversible co-processor has to uncompute the process of obtaining gradients to clean up the gradient tape. The whole process of obtaining gradients does not have a lower bound of energy consumption.

We still face challenges to support reversible hardwares. One of the most challenging one is arithmetic instructions should be redesigned to support better reversible programs. The major obstacle to exact reversibility programming is the current floating-point adders and multipliers used in our computing devices are not exactly reversible. 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). Alternatives include fixed point numbers Fix and logarithmic numbers Taylor *et al.* (1988); Log, where logarithmic number system is reversible under `*=` and `/=` but not addition and subtraction. We also need instructions like `comefrom` as a partener of `goto`. Many people know `comefrom` from the joke Wikipedia contributors (2020b) to complaint people who use `goto` frequently. It turns out to be necessary for compiling a reversible program.

Reversible instructions could be executed on energy-efficient reversible hardware. In the introduction, we mentioned several reversible hardware. Reversible hardware can be devices

supporting reversible gates like the Toffoli gate and the Fredkin gate, or devices like an adiabatic CMOS with the ability to recover signal energy. The latter is known as the generalized reversible computing Frank (2005); Frank (2017b). It is already a better choice as the computing device in a spacecraft DeBenedictis *et al.* (2017). Since reversible programming is an exceptional platform for differential programming, building an energy-efficient artificial intelligence (AI) coprocessors would be also a promising direction.

The development of reversible compiling theory can benefit quantum compiling Chong *et al.* (2017) directly, as it bridges classical computing and quantum computing. Building a universal quantum computer Nielsen and Chuang (2002) is difficult. The difficulty lies in the fact that it is hard to protect a quantum state. 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. It is technically more 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 CPHASE quantum gate Ruiz-Perez and Garcia-Escartin (2017). Currently, most quantum programming language preassumes a classical coprocessor and uses classical control flows Svore *et al.* (2018) in universal quantum computing. However, we believe reversible compiling technologies, including reversible control flows, are also very important to a universal quantum computer.

6.3 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 going to be deallocated. As a result, even if an ancilla can be uncomputed rigorously in the original program, its `GVar` wrapped version is not necessarily safely deallocated. 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 harm the reversibility of a function.*

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 indenpedent of input $\mathbf{x}^{i-1}$, we have

$$\frac{\partial b}{\partial \mathbf{x}^{i-1}} = \mathbf{0}. \tag{5}$$

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

$$\frac{\partial \mathbf{x}^L}{\partial \mathbf{x}^{i-1}} = \frac{\partial \mathbf{x}^L}{\partial \mathbf{x}^i}\frac{\partial \mathbf{x}^i}{\partial \mathbf{x}^{i-1}} + \frac{\partial \mathbf{x}^L}{\partial b}\frac{\partial b}{\partial \mathbf{x}^{i-1}}, \tag{6}$$

where the second term with $\frac{\partial \mathbf{x}^L}{\partial b}$ vanishes naturally. □

6.4 Shared read and write problem

Let's first consider 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 instuction can be easily identified by the compiler easily. However, it becomes tricky when one runs the program in a parallel way. For example, in CUDA programming, every thread may want to write to the same gradient field of a scalar. How to solve the shared write in CUDA programming is still an open problem, which limits the power of AD on GPU.

6.5 Outlook

We can use NiLang to solve many existing issues related to AD. We can use it to generate AD rules for existing machine learning packages like ReverseDiff Rev, Zygote Innes *et al.* (2019), KNet KNe, 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) with NiLang. 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. We show the potential of self-consistent training in Appendix D.

To solve the above problems better, people can improve reversible programming from multiple perspectives. First, we need a better compiler suited for compiling reversible programs. It can decrease the uncomputing overheads automatically for us. A better compiler can also help to avoid the problem of shared memory write problem on GPU when computing gradients. Then, we need a number system to avoid rounding errors. Currently, we can simulate rigorous reversible arithmetics with the fixed-point number package Fix; Log. A more efficient fixed point or log number

operations requires instruction-level design. Finally, the improvement from the hardware level will arm reversible differential programming with energy efficiency, which is also very important to help variational programming to solve practical issues better. For example, we can build an energy-efficient AI chip in our cellular phone with reversible computing devices. These improvements need the participation of people from multiple fields.

7 Acknowledgments

Jin-Guo Liu thank Lei Wang for motivating the project with possible applications to reversible integrator, normalizing flow, and neural ODE. Johann-Tobias Schäg for deepening the discussion about reversible programming with his mathematicians head. Marisa Kiresame and Xiu-Zhe Luo for discussion on the implementation details of source-to-source automatic differentiation, Shuo-Hui Li for helpful discussion on differential geometry, Tong Liu and An-Qi Chen for helpful discussion on quantum adders and multipliers, Ying-Bo Ma for correcting typos by submitting pull requests, Chris Rackauckas for helpful discussion on reversible integrator, Mike Innes for reviewing the comments about Zygote, Jun Takahashi for discussion about the graph embedding problem, Simon Byrne and Chen Zhao for helpful discussion on floating-point and logarithmic numbers. The authors are supported by the National Natural Science Foundation of China under Grant No. 11774398, the Strategic Priority Research Program of Chinese Academy of Sciences Grant No. XDB28000000.

References

L. Hascoet and V. Pascual, ACM Transactions on Mathematical Software (TOMS) **39**, 20 (2013).

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, in *NIPS Autodiff Workshop* (2017).

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] .

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," (2015), software available from tensorflow.org.

J.-G. L. Hao Xie and L. Wang, arXiv:2001.04121 .

G. H. Golub and C. F. Van Loan, *Matrix computations*, Vol. 3 (JHU press, 2012).

H.-J. Liao, J.-G. Liu, L. Wang, and T. Xiang, Physical Review X **9** (2019), 10.1103/physrevx.9.031041.

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] .

M. Seeger, A. Hetzel, Z. Dai, E. Meissner, and N. D. Lawrence, "Auto-differentiating linear algebra," (2017), arXiv:1710.08717 [cs.MS] .

Z.-Q. Wan and S.-X. Zhang, "Automatic differentiation for complex valued svd," (2019), arXiv:1909.02659 [math.NA] .

C. Hubig, "Use and implementation of autodifferentiation in tensor network methods with complex scalars," (2019), arXiv:1907.13422 [cond-mat.str-el] .

"ReverseDiff.jl," `https://github.com/JuliaDiff/ReverseDiff.jl`.

M. Innes, "Don't unroll adjoint: Differentiating ssa-form programs," (2018), arXiv:1810.07951 [cs.PL] .

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

Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," (2015), arXiv:1506.00019 [cs.LG] .

K. He, X. Zhang, S. Ren, and J. Sun, 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016), 10.1109/cvpr.2016.90.

M. MacKay, P. Vicol, J. Ba, and R. B. Grosse, 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. 9029–9040.

D. Maclaurin, D. Duvenaud, and R. Adams, in *Proceedings of the 32nd International Conference on Machine Learning*, Proceedings of Machine Learning Research, Vol. 37, edited by F. Bach and D. Blei (PMLR, Lille, France, 2015) pp. 2113–2122.

J. Behrmann, D. Duvenaud, and J. Jacobsen, CoRR **abs/1811.00995** (2018), arXiv:1811.00995 .

A. N. Gomez, M. Ren, R. Urtasun, and R. B. Grosse, in *Advances in Neural Information Processing Systems 30*, edited by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Curran Associates, Inc., 2017) pp. 2214–2224.

J.-H. Jacobsen, A. W. Smeulders, and E. Oyallon, in *International Conference on Learning Representations* (2018).

J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, arXiv preprint arXiv:1209.5145 (2012).

J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, SIAM Review **59**, 65–98 (2017).

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

M. P. Frank, IEEE Spectrum **54**, 32–37 (2017a).

"MLStyle.jl," `https://github.com/thautwarm/MLStyle.jl`.

C. Lutz, "Janus: a time-reversible language," (1986), *Letter to R. Landauer*.

M. P. Frank, *The R programming language and compiler*, Tech. Rep. (MIT Reversible Computing Project Memo, 1997).

I. Lanese, N. Nishida, A. Palacios, and G. Vidal, Journal of Logical and Algebraic Methods in Programming **100**, 71–97 (2018).

T. Haulund, "Design and implementation of a reversible object-oriented programming language," (2017), arXiv:1707.07845 [cs.PL] .

M. P. Frank and T. F. Knight Jr, *Reversibility for efficient computing*, Ph.D. thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and ... (1999).

J. G. Koller and W. C. Athas, in *Workshop on Physics and Computation* (1992) pp. 267–270.

R. C. Merkle, R. A. Freitas, T. Hogg, T. E. Moore, M. S. Moses, and J. Ryley, Journal of Mechanisms and Robotics **10** (2018), 10.1115/1.4041209.

K. Likharev, IEEE Transactions on Magnetics **13**, 242 (1977).

V. K. Semenov, G. V. Danilov, and D. V. Averin, IEEE Transactions on Applied Superconductivity **13**, 938 (2003).

R. Landauer, IBM journal of research and development **5**, 183 (1961).

I. Hänninen, G. Snider, and C. Lent, "Adiabatic cmos: Limits of reversible energy recovery and first steps for design automation," (2014) pp. 1–20.

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

D. R. Jefferson, ACM Transactions on Programming Languages and Systems (TOPLAS) **7**, 404 (1985).

B. Boothe, ACM SIGPLAN Notices **35**, 299 (2000).

C. H. Bennett (1973).

Wikipedia contributors, "Einstein notation — Wikipedia, the free encyclopedia," (2020a), [Online; accessed 20-February-2020].

J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in julia," (2016), arXiv:1607.07892 [cs.MS] .

A. McInerney, *First steps in differential geometry* (Springer, 2015).

M. Arjovsky, A. Shah, and Y. Bengio, CoRR **abs/1511.06464** (2015), arXiv:1511.06464 .

S. Wisdom, T. Powers, J. R. Hershey, J. L. Roux, and L. Atlas, "Full-capacity unitary recurrent neural networks," (2016), arXiv:1611.00035 [stat.ML] .

L. Jing, Y. Shen, T. Dubcek, J. Peurifoy, S. A. Skirlo, M. Tegmark, and M. Soljacic, CoRR **abs/1612.05231** (2016), arXiv:1612.05231 .

C.-K. LI, R. ROBERTS, and X. YIN, International Journal of Quantum Information **11**, 1350015 (2013).

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 .

F. Srajer, Z. Kukelova, and A. Fitzgibbon, Optimization Methods and Software **33**, 889 (2018).

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 .

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

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] .

"FixedPointNumbers.jl," `https://github.com/JuliaMath/FixedPointNumbers.jl`.

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

"LogarithmicNumbers.jl," `https://github.com/cjdoris/LogarithmicNumbers.jl`.

Wikipedia contributors, "Comefrom — Wikipedia, the free encyclopedia," (2020b), [Online; accessed 8-March-2020].

M. P. Frank, in *35th International Symposium on Multiple-Valued Logic (ISMVL'05)* (2005) pp. 168–185.

M. P. Frank, in *Reversible Computation*, edited by I. Phillips and H. Rahaman (Springer International Publishing, Cham, 2017) pp. 19–34.

F. T. Chong, D. Franklin, and M. Martonosi, Nature **549**, 180 (2017).

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.

"KNet.jl," https://github.com/denizyuret/Knet.jl.

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. Bender, P.-H. Heenen, and P.-G. Reinhard, Rev. Mod. Phys. **75**, 121 (2003).

T. Besard, C. Foket, and B. D. Sutter, CoRR **abs/1712.03112** (2017), arXiv:1712.03112 .

"KernelAbstractions.jl," https://github.com/JuliaGPU/KernelAbstractions.jl.

A NiLang Grammar

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

```
"""
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.

⟨Stmts⟩ ::= ϵ
| ⟨Stmt⟩
| ⟨Stmts⟩ ⟨Stmt⟩

⟨Stmt⟩ ::= ⟨BlockStmt⟩
| ⟨IfStmt⟩
| ⟨WhileStmt⟩
| ⟨ForStmt⟩
| ⟨InstrStmt⟩
| ⟨RevStmt⟩
| ⟨AncillaStmt⟩
| ⟨TypecastStmt⟩
| ⟨@routine⟩ ⟨Stmt⟩
| ⟨@safe⟩ *JuliaExpr*
| ⟨CallStmt⟩

⟨BlockStmt⟩ ::= *begin* ⟨Stmts⟩ *end*

⟨RevCond⟩ ::= *(JuliaExpr , JuliaExpr)*

⟨IfStmt⟩ ::= *if* ⟨RevCond⟩ ⟨Stmts⟩ [*else* ⟨Stmts⟩] *end*

⟨WhileStmt⟩ ::= *while* ⟨RevCond⟩ ⟨Stmts⟩ *end*

⟨Range⟩ ::= *JuliaExpr : JuliaExpr* [*: JuliaExpr*]

⟨ForStmt⟩ ::= *for ident* = ⟨Range⟩ ⟨Stmts⟩ *end*

⟨KwArg⟩ ::= *ident* = *JuliaExpr*

⟨KwArgs⟩ ::= [⟨KwArgs⟩ ,] ⟨KwArg⟩

⟨CallStmt⟩ ::= *JuliaExpr (* [⟨DataViews⟩] [*;* ⟨KwArgs⟩] *)*

⟨Constant⟩ ::= *num* | π | *true* | *false*

⟨InstrBinOp⟩ ::= += | -= | $\veebar$=

⟨InstrTrailer⟩ ::= [.] *(* [⟨DataViews⟩] *)*

⟨InstrStmt⟩ ::= ⟨DataView⟩ ⟨InstrBinOp⟩ *ident* [⟨InstrTrailer⟩]

⟨RevStmt⟩ ::= ~ ⟨Stmt⟩

⟨AncillaStmt⟩ ::= *ident* ← *JuliaExpr*
| *ident* → *JuliaExpr*

⟨TypecastStmt⟩ ::= *(JuliaExpr => JuliaExpr) (ident)*

⟨@routine⟩ ::= *@routine ident* ⟨Stmt⟩

⟨@safe⟩ ::= *@safe JuliaExpr*

⟨DataViews⟩ ::= ϵ
| ⟨DataView⟩
| ⟨DataViews⟩ , ⟨DataView⟩
| ⟨DataViews⟩ , ⟨DataView⟩ ...

⟨DataView⟩ ::= ⟨DataView⟩ *[JuliaExpr]*
| ⟨DataView⟩ . *ident*
| *JuliaExpr (* ⟨DataView⟩ *)*
| ⟨DataView⟩ ′
| - ⟨DataView⟩
| ⟨Constant⟩
| *ident*

B Instructions and Backward Rules

instruction	translated	symbol
y += $f(args...)$	`PlusEq(f)(args...)`	$\oplus(f)$
y −= $f(args...)$	`MinusEq(f)(args...)`	$\ominus(f)$
$y \veebar= f(args...)$	`XorEq(f)(args...)`	$\odot(f)$

Table 7: Instructions, the functions that they compiled to, and their symbolic representations.

The list of instructions implemented in NiLang

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 += $a * b$	$y + a * b, a, b$
y += a/b	$y + a/b, a, b$
y += a^b	$y + a^b, a, b$
y += identity(x)	$y + x, x$
y += exp(x)	$y + e^x, x$
y += log(x)	$y + \log x, x$
y += sin(x)	$y + \sin x, x$
y += cos(x)	$y + \cos x, x$
y += abs(x)	$y + \|x\|, x$
NEG(y)	$-y$
CONJ(y)	y'

Table 8: Predefined reversible instructions in NiLang.

C 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 A.

C.1 Backward rules for instructions

For function $\vec{y} = f(\vec{x})$, its Jacobian is $J_{ij} = \frac{\partial y_i}{\partial x_j}$ and its Hessian is $H_{ij}^k = \frac{\partial y_k}{x_i x_j}$. We have the following local Jacobians and Hessians on the above instructions.

1. $a \mathrel{+}= b$

$$J = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$
$$H = \mathbf{0}$$

The inverse is $a \mathrel{-}= b$, and its Jacobian is the inverse of the matrix above.

$$J(f^{-1}) = J^{-1} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$$

In the following, we omit the Jacobians and Hessians of inverse functions.

2. $a \mathrel{+}= b * c$

$$J = \begin{pmatrix} 1 & c & b \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$
$$H^a_{bc} = H^a_{cb} = 1, else\ 0$$

3. $a \mathrel{+}= b/c$

$$J = \begin{pmatrix} 1 & 1/c & -b/c^2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$
$$H^a_{cc} = 2b/c^3,$$
$$H^a_{bc} = H^a_{cb} = -1/c^2, else\ 0$$

4. $a \mathrel{+}= b^c$

$$J = \begin{pmatrix} 1 & cb^{c-1} & b^c \log b \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$
$$H^a_{bc} = H^a_{cb} = b^{c-1} + cb^{c-1} \log b,$$
$$H^a_{bb} = (c-1)cb^{c-2},$$
$$H^a_{cc} = b^c \log^2 b, else\ 0$$

5. $a \mathrel{+}= e^b$

$$J = \begin{pmatrix} 1 & e^b \\ 0 & 1 \end{pmatrix}$$
$$H^a_{bb} = e^b, else\ 0$$

6. $a \mathrel{+}= \log b$

$$J = \begin{pmatrix} 1 & 1/b \\ 0 & 1 \end{pmatrix}$$
$$H^a_{bb} = -1/b^2, else\ 0$$

7. $a \mathrel{+}= \sin b$

$$J = \begin{pmatrix} 1 & \cos b \\ 0 & 1 \end{pmatrix}$$
$$H^a_{bb} = -\sin b, else\ 0$$

8. $a \mathrel{+}= \cos b$

$$J = \begin{pmatrix} 1 & -\sin b \\ 0 & 1 \end{pmatrix}$$
$$H^a_{bb} = -\cos b, else\ 0$$

9. $a \mathrel{+}= |b|$

$$J = \begin{pmatrix} 1 & \operatorname{sign}(b) \\ 0 & 1 \end{pmatrix}$$
$$H = \mathbf{0}$$

10. $a = -a$

$$J = \begin{pmatrix} -1 \end{pmatrix}$$
$$H = \mathbf{0}$$

11. $\mathrm{SWAP}(a, b) = (b, a)$

$$J = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$
$$H = \mathbf{0}$$

12.

$$\mathrm{ROT}(a, b, \theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}$$

$$J = \begin{pmatrix} \cos\theta & -\sin\theta & -b\cos\theta - a\sin\theta \\ \sin\theta & \cos\theta & a\cos\theta - b\sin\theta \\ 0 & 0 & 1 \end{pmatrix}$$
$$H^a_{a\theta} = H^a_{\theta,a} = -\sin\theta,$$
$$H^a_{b\theta} = H^a_{\theta,b} = -\cos\theta,$$
$$H^a_{\theta\theta} = -a\cos\theta + b\sin\theta,$$
$$H^b_{a\theta} = H^b_{\theta a} = \cos\theta,$$
$$H^b_{b\theta} = H^b_{\theta b} = -\sin\theta,$$
$$H^b_{\theta\theta} = -b\cos\theta - a\sin\theta, else\ 0$$

D Learn by consistency

Consider a training that with input $\mathbf{x}^*$ and output $\mathbf{y}^*$, find a set of parameters $\mathbf{p}_x$ that satisfy $\mathbf{y}^* = f(\mathbf{x}^*, \mathbf{p}_x)$. In traditional machine learning, we define a loss $\mathcal{L} = \text{dist}(\mathbf{y}^*, f(\mathbf{x}^*, \mathbf{p}_x))$ and minimize it with gradient $\frac{\partial L}{\partial \mathbf{p}_x}$. This works only when the target function is locally differentiable.

Here we provide an alternative by making use of reversibility. We construct a reversible program $\mathbf{y}, \mathbf{p}_y = f_r(\mathbf{x}, \mathbf{p}_x)$, where $\mathbf{p}_x$ and $\mathbf{p}_y$ are "parameter" spaces on the input side and output side. The algorithm can be summarized as

Algorithm 2: Learn by consistency

Result: $\mathbf{p}_x$
Initialize $\mathbf{x}$ to $\mathbf{x}^*$, parameter space $\mathbf{p}_x$ to random.
if $\mathbf{p}_y$ *is* `null` **then**
  $\mathbf{x}, \mathbf{p}_x = f_r^{-1}(\mathbf{y}^*)$
else
  $\mathbf{y}, \mathbf{p}_y = f_r(\mathbf{x}, \mathbf{p}_x)$
  while $\mathbf{y} \not\approx \mathbf{y}^*$ **do**
    $\mathbf{y} = \mathbf{y}^*$
    $\mathbf{x}, \mathbf{p}_x = f_r^{-1}(\mathbf{y}, \mathbf{p}_y)$.
    $\mathbf{x} = \mathbf{x}^*$
    $\mathbf{y}, \mathbf{p}_y = f_r(\mathbf{x}, \mathbf{p}_x)$

Here, `parameter`$(\cdot)$ is a function for taking the parameter space. This algorithm utilizes the self-consistency relation

$$\mathbf{p}_x^* = \texttt{parameter}(f_r^{-1}(\mathbf{y}^*, \texttt{parameter}(f_r(\mathbf{x}^*, \mathbf{p}_x^*)))), \tag{7}$$

A similar idea of training by consistency is used in self-consistent mean-field theory Bender *et al.* (2003) in physics. Finding the self-consistent relation is crucial to self-consistency based training. Here, the reversibility provides a natural self-consistency relation. However, it is not a silver bullet; let's consider the following example.

```
@i function f1(y!, x, p!)
    p! += identity(x)
    y! -= exp(x)
    y! += exp(p!)
end

@i function f2(y!, x!, p!)
    p! += identity(x!)
    y! -= exp(x!)
    x! -= log(-y!)
    y! += exp(p!)
end
```

```
function train(f)
    loss = Float64[]
    p = 1.6
    for i=1:100
        y!, x = 0.0, 0.3
        @instr f(y!, x, p)
        push!(loss, y!)
        y! = 1.0
        @instr (~f)(y!, x, p)
    end
    loss
end
```

Functions `f1` and `f2` computes $f(x, p) = e^{(p+x)} - e^x$ and stores the output in a new memory `y!`. The only difference is `f2` uncomputes x arithmetically. The task of the training is to find a p that makes the output value equal to the target value 1. After 100 steps, `f2` runs into the fixed point with x equal to 1 upto machine precision. However, parameters in `f1` does not change at all. The training of `f1` fails because this function actually computes $\texttt{f1}(y, x, p) = y + e^{(p+x)} - e^x, x, x + p$, where the training parameter p is completely determined by the parameter space on the output side $x \cup x + p$. As a result, shifting y directly is the only approach to satisfy the consistency relation. On the other side, $\texttt{f2}(y, x, p) = y + e^{(p+x)} - e^x, \tilde{0}, x + p$, the output parameters $\tilde{0} \cup x + p$ can not uniquely determine input parameters p and x. Here, we use $\tilde{0}$ to denote the zero with rounding error.

By viewing $\mathbf{x}$ and parameters in $\mathbf{p}_x$ as variables, we can study the trainability from the information perspective.

***Theorem* 2.** *Only if the the conditional entropy $S(\mathbf{y}|\mathbf{p}_y)$ is nonzero, algorithm 2 is trainable.*

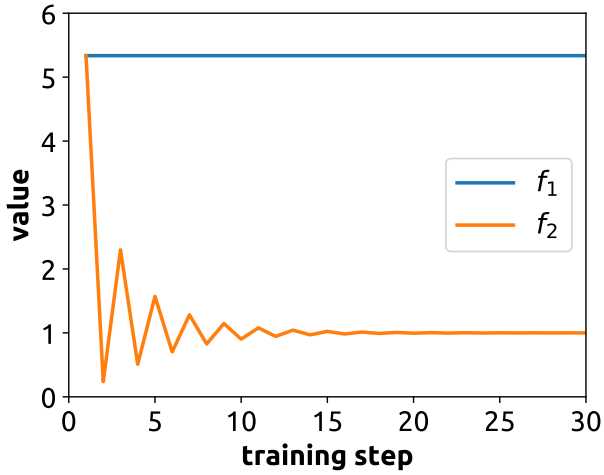


Figure 7: The output value `y!` as a function of self-consistent training step.

Proof. The above example reveals a fact that training is impossible when output parameters completely determines input parameters (or $S(\mathbf{p}_x|\mathbf{p}_y) = 0$).

$$\begin{aligned} S(\mathbf{p}_x|\mathbf{p}_y) &= S(\mathbf{p}_x \cup \mathbf{p}_y) - S(\mathbf{p}_y) \\ &\leq S\left((\mathbf{p}_x \cup \mathbf{x}) \cup \mathbf{p}_y\right) - S(\mathbf{p}_y), \\ &\leq S\left((\mathbf{p}_y \cup \mathbf{y}) \cup \mathbf{p}_y\right) - S(\mathbf{p}_y), \\ &\leq S(\mathbf{y}|\mathbf{p}_y). \end{aligned} \tag{8}$$

The third line uses the bijectivity $S(\mathbf{x} \cup \mathbf{p}_x) = S(\mathbf{y} \cup \mathbf{p}_y)$. This inequality shows that when $S(\mathbf{y}|\mathbf{p}_y) = 0$, i.e., the output parameters contain all information in output, the input parameters are entirely determined and the training can not work. □

In the above example, it corresponds to the case $S\left(e^{(x+y)-e^x}|x \cup x+y\right) = 0$ in `f1`. The solution is to remove the information redundancy in output parameter space through uncomputing, as shown in `f2`. Besides, the Fibonacci example is often used in a reversible language as a tutorial, NiLang implementation could be found in Appendix G.

E Functions used in the main text

We list the functions used in Sec. 4 as bellow.

```
"""
get the summation of an array.
"""
@i function isum(out!, x::AbstractArray)
    for i=1:length(x)
        out! += identity(x[i])
    end
end

"""
computing factorial.
"""
@i function ifactorial(out!, n)
    out! += identity(1)
    for i=1:n
        mulint(out!, i)
    end
end

"""
dot product.
"""
@i function dot(out!, v1::Vector{T}, v2) where T
    for i = 1:length(v1)
        out! += v1[i]'*v2[i]
    end
end

"""
```

```
squared norm.
"""
@i function norm2(out!, vec::Vector{T}) where T
    anc1 ← zero(T)
    for i = 1:length(vec)
        anc1 += identity(vec[i]')
        out! += anc1*vec[i]
        anc1 -= identity(vec[i]')
    end
end

"""
Variance and mean value from squared values `sqv
    `.
"""
@i function var_and_mean_sq(var!, mean!, sqv)
    sqmean ← zero(mean!)
    @inbounds for i=1:length(sqv)
        mean! += sqv[i] ^ 0.5
        var! += identity(sqv[i])
    end
    divint(mean!, length(sqv))
    divint(var!, length(sqv))
    sqmean += mean! ^ 2
    var! -= identity(sqmean)
    sqmean -= mean! ^ 2
    mulint(var!, length(sqv))
    divint(var!, length(sqv)-1)
end
```

F CUDA compitibility

CUDA programming is playing a more and more significant role in high-performance computing. In Julia, one can write kernel functions in native Julia language with CUDAnative Besard *et al.* (2017). NiLang is compatible with CUDAnative and KernelAbstractions Ker, and one can write a reversible kernel like the following.

```
using CuArrays, CUDAnative, GPUArrays
using NiLang, NiLang.AD

@i @inline function swap_kernel(state, mask1, mask2)
    @invcheckoff begin
        b ← (blockIdx().x-1) *
            blockDim().x + threadIdx().x - 1
        if (b < length(state), ~)
            if (b&mask1==0 && b&mask2==mask2, ~)
                SWAP(state[b+1], state[b ⊻
                    (mask1|mask2) + 1])
            end
        end
        b → (blockIdx().x-1) *
            blockDim().x + threadIdx().x - 1
    end
end
```

This kernel function simulates the SWAP gate in quantum computing. Here, one must use the macro `@invcheckoff` to turn off the reversibility checks. It is necessary because the possible error thrown in a kernel function can not be handled on a CUDA kernel. One can launch this kernel function to GPUs with a single macro `@cuda`, as shown in the following using case.

```
julia> @i function instruct!(state::CuVector,
            gate::Val{:SWAP}, locs::Tuple{Int,
    Int})
          mask1 ← 1 << (tget(locs, 1)-1)
          mask2 ← 1 << (tget(locs, 2)-1)
          XY ← GPUArrays.
    thread_blocks_heuristic(
                length(state))
          @cuda threads=tget(XY,1) blocks=tget(
    XY,
                2) swap_kernel(state, mask1,
    mask2)
      end

julia> instruct!(CuArray(randn(8)),
           Val(:SWAP), (1,3))[1]
8-element CuArray{Float64,1,Nothing}:
 -0.06956048379200473
 -0.6464176838567472
 -0.06523362834285944
 -0.7314356941903547
  1.512329204247244
  0.9773772766637732
  1.6473223915215722
 -1.0631789613639087
```

One can also write kernels with KernelAbstaction. It solves many compatibility issues related to different function calls on GPU and CPU.

```
@i @kernel function swap_kernel2(state, mask1, mask2)
    @invcheckoff begin
        b ← @index(Global)
        if (b < length(state), ~)
            if (b&mask1==0 && b&mask2==mask2, ~)
                SWAP(state[b+1], state[b ⊻
                    (mask1|mask2) + 1])
            end
        end
        b → @index(Global)
    end
end
```

We can use the macro `@launchkernel` to launch a kernel. The first parameter is a device. The second parameter is the block size. The third parameter is the number of threads. The last parameter is a kernel function call to be launched.

```
julia> @i function instruct!(state::CuVector,
            gate::Val{:SWAP}, locs::Tuple{Int,
    Int})
          mask1 ← 1 << (tget(locs, 1)-1)
          mask2 ← 1 << (tget(locs, 2)-1)
          XY ← GPUArrays.
    thread_blocks_heuristic(
                length(state))
          @launchkernel CUDA() 256 length(out!
                ) swap_kernel2(state, mask1,
    mask2)
      end

julia> instruct!(CuArray(randn(8)),
           Val(:SWAP), (1,3))[1]
8-element CuArray{Float64,1,Nothing}:
  2.1492759883720525
  2.326837084303501
  1.4587667131427016
 -1.3273806428138293
 -0.03975355575683114
 -0.10763082744447787
 -1.7111718557581195
 -0.47922613687722704
```

G Computing Fibonacci Numbers

The following is an example that everyone likes, computing Fibonacci number recursively.

```
using NiLang

@i function rfib(out!, n::T) where T
    n1 ← zero(T)
    n2 ← zero(T)
    @routine begin
        n1 += identity(n)
        n1 -= identity(1)
        n2 += identity(n)
        n2 -= identity(2)
    end
    if (value(n) <= 2, ~)
        out! += identity(1)
    else
        rfib(out!, n1)
        rfib(out!, n2)
    end
    ~@routine
end
```

The time complexity of this recursive algorithm is exponential to input `n`. It is also possible to write a reversible linear time with for loops. A slightly non-trivial task is computing the first Fibonacci number that greater or equal to a certain number z, where a `while` statement is required.

```
@i function rfibn(n!, z)
    @safe @assert n! == 0
    out ← 0
    rfib(out, n!)
    while (out < z, n! != 0)
        ~rfib(out, n!)
        n! += identity(1)
        rfib(out, n!)
    end
    ~rfib(out, n!)
end
```

In this example, the postcondition `n!=0` in the `while` statement is false before entering the loop, and it becomes true in later iterations. In the reverse program, the `while` statement stops at `n==0`. If executed correctly, a user will see the following result.

```
julia> rfib(0, 10)
(55, 10)

julia> rfibn(0, 100)
(12, 100)

julia> (~rfibn)(rfibn(0, 100)...)
(0, 100)
```

This example shows how an addition postcondition provided by the user can help to reverse a control flow without caching controls.