Differentiate Everything with a Reversible Domain-Specific Language

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

Reverse-mode automatic differentiation (AD) suffers from the issue of having too much space overhead to trace back intermediate computational states for backpropagation. The traditional method to trace back states is called checkpointing that stores intermediate states into a global stack and restore state through either stack pop or re-computing. The overhead of stack manipulations and re-computing makes the general purposed (not tensor-based) AD engines unable to meet many industrial needs. Instead of checkpointing, we propose to use reverse computing to trace back states by designing and implementing a reversible programming eDSL, where a program can be executed bi-directionally without implicit stack operations. The absence of implicit stack operations makes the program compatible with existing compiler features, including utilizing existing optimization passes and compiling the code as GPU kernels. We implement AD for sparse matrix operations and some machine learning applications to show that the performance of our framework has state-of-the-art performance.

1 Introduction

Most popular autmatic differetiation (AD) packages in the market, such as TensorFlow Abadi *et al.* (2015), Pytorch Paszke *et al.* (2017) and Flux Innes *et al.* (2018) implements reverse mode AD at the tensor level to meet the need of machine learning. Later, People in the scientific computing domain also find the powerfulness of these AD tools, people use these them to solve scientific problems such as seismic inversion Zhu *et al.* (2020), variational quantum circuits simulation Bergholm *et al.* (2018) and variational tensor network simulation Liao *et al.* (2019); Roberts *et al.* (2019). To meet the diversed need in these applications, one often needs to add manually defined backward rules, for example

1. In order to differentiate sparse matrix operations for Hamiltonian engineering Hao Xie and Wang, we need to define backward rules for sparse matrix operations and dominant eigensolvers Golub and Van Loan (2012),
2. In tensor network algorithms to study the phase transition problem Golub and Van Loan (2012); Liao *et al.* (2019); Seeger *et al.* (2017); Wan and Zhang (2019); Hubig (2019), one needs defining backward rules for singular value decomposition (SVD) function and QR decomposition.

To avoid defining backward rules manually, one can also use a general purposed AD (GP-AD) packages like Tapenade Hascoet and Pascual (2013), OpenAD Utke *et al.* (2008) and Zygote Innes (2018); Innes *et al.* (2019) to differentiate a general program. These tools has been used in non-tensor based applications such as bundle adjustment Shen and Dai (2018) and earth system simulation Forget *et al.* (2015). They read the source code from a user and generate code to compute the gradients. However, these packages have their own limitations too. In many practical applications, differentiating a program might do billions of computations. Frequent caching of data slows down the program significantly, while the memory usage will become a bottleneck as well. Moreover, implicit caching is not compatible with differentiating GPU kernels.

These needs call for a GP-AD framework that does not cache for users automatically. Hence we propose to implement the reverse mode AD on a reversible (domain-specific) programming language Perumalla (2013); Frank (2017). So that the intermediate states of a program can be traced backward with no extra effort. In a reversible languages, the memory allocation and deallocation are explicit, with which flexible time-space tradeoff is available. Meanwhile, one can also utilize the reversibility to reverse the program without space overhead, which is proven to significantly decrease the memory usage in unitary recurrent neural networks MacKay *et al.* (2018), normalizing flow Dinh *et al.* (2014), hyperparameter learning Maclaurin *et al.* (2015) and residual neural networks Behrmann *et al.* (2018). Using reversible programming language makes all these happen naturally without extra framework designs.

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 to study reversible programming is to support reversible computing 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), where a reversible computing device is more energy-efficient by the Landauer's principle Landauer (1961). The above reversible programming languages are well defined so that can be compiled to future hardwares. However, the drawback is they can hardly be used in real scientific programming, since most of them do not have basic elements like floating point numbers, arrays and complex numbers. This motivates us to build a new embeded domain specific language (eDSL) in Julia Bezanson *et al.* (2012, 2017).

In this paper, we first introduce the language design of a reversible programming language and introduce our reversible eDSL NiLang in Sec. 2. In Sec. 3, we explain the implementation of automatic differentiation in this eDSL. In Sec. 4, we benchmark the performance of NiLang with other AD packages and explain why reversible programming AD is fast. In the appendix, we show the detailed language design of NiLang, show some examples used in the benchmark, discuss several important issues including the time-space tradeoff, reversible instructions and hardware, and finally, an outlook to some open problems to be solved.

2 Language design

In the introduction, we have introduced several reversible programming languages. These languages lacks essential components for scientific programming like arrays and complex numbers, while none of them can be embeded in other machine learning frameworks easily. Hence we developped an embedded domain-specific language (eDSL) NiLang on top of the host language language Julia Bezanson *et al.* (2012, 2017). Julia is a popular language for scientific programming and machine learning. We choose Julia mainly for speed. Julia is a language with high abstraction, however, its clever design of type inference and just in time compiling make it has a C like speed. Also, it has rich features for meta-programming, and the package for pattern matching MLStyle allows us to define an eDSL compiler in less than 2000 lines. Comparing with a regular reversible programming language, NiLang features array operations, rich number systems including floating point number, complex number, fixed point number and logarithmic number. Besides the above nice features, it also has some "bad" features to meet the practical needs. For example, it views the floating point $+$ and $-$ operations as reversible. It also allows user to extend instruction sets and sometimes inserting external statements. These features are not compatible with future reversible hardwares. By the time of writting, the version of NiLang is v0.7.2.

2.1 Reversible functions and arithmetic instructions

Mathematically, a irreversilbe mapping `y = f(args...)` can be trivially transfromed to its reversible form `y += f(args...)` or `y ⊻= f(args...)` ($\veebar$ is the bitwise `XOR`), where `y` is a pre-emptied variable. But in numeric computing with finite precision, this is not the whole story, because the reversibility of arithmetic instruction is closely related to the number system. For integer and fixed point number system, `y += f(args...)` and `y -= f(args...)` are rigorously reversible. For logarithmic number system and tropical number system Speyer and Sturmfels (2009), `y *= f(args...)` and `y /= f(args...)` as reversible (not introducing the zero element). While for floating point numbers, none of the above operations are regorously reversible. However, for convenience, we ignore the rounding errors in floating point `+` and `-` operations and treat them on equal footing with fixed point numbers in the following discussion. Besides the above operations, `SWAP` operation that exchanges the values of two variables is also widely used in reversible programming.

The following code defines a reversible multiplier.

Listing 1: A reversible mutiplier

```
julia> using NiLang

julia> @i function multiplier(y!::Real, a::Real, b::Real)
           y! += a * b
       end

julia> multiplier(2, 3, 5)
(17, 3, 5)

julia> (~multiplier)(17, 3, 5)
(2, 3, 5)
```

Macro `@i` generates two functions that are reversible to each other `multiplier` and `~multiplier`, each defines a mapping $\mathbb{R}^3 \to \mathbb{R}^3$. The ! after a symbol is a part of the name, as a convension to indicate that this variable is changed.

2.2 Reversible memory management

A distinct feature of reversible memory management is that the content of a variable must be known when it is deallocated. We denote the allocation of a zero emptied memory as `x ← 0`, and the corresponding deallocation as `x → 0`. A variable can also be pushed to a stack and used later with a pop statement. If a variable is allocated and deallocated in a local scope, we call it an ancilla.

Listing 2: Reversible complex valued log function $y += \log(|x|) + i\mathrm{Arg}(x)$.

```
@i @inline function (:+=)(log)(y!::Complex{T
    }, x::Complex{T}) where T
    n ← zero(T)
    n += abs(x)

    y!.re += log(n)
    y!.im += angle(x)

    n -= abs(x)
    n → zero(T)
end
```

Listing 3: Compute-copy-uncompute version of Listing. 2

```
@i @inline 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
```

Listing. 2 defines the complex valued accumulative log function. The macro `@inline` tells the compiler that this function can be inlined. One can input "←" and "→" by typing "\leftarrow[TAB KEY]" and "\rightarrow[TAB KEY]" respectively in a Julia editor or REPL. NiLang does not have immutable structs, so that the real part `y!.re` and imaginary `y!.im` of a complex number can be changed directly. It is easy to verify that the bottom two lines in the function body are the reverse of the top two lines. We call the bottom two lines *uncomputes* the top two lines. The motivation is to zero clear the contents in ancilla `n` so that it can be deallocated correctly. *Compute-copy-uncompute* is a useful design pattern in reversible programming so that we created a pair of macros `@routine` and `~@routine` for it. One can rewrite the above function as in Listing. 3.

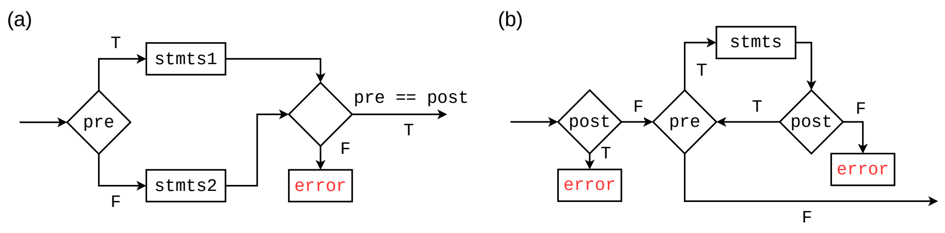

Figure 1: The flow chart for reversible (a) `if` statement and (b) `while` statement. “pre” and “post” represents precondition and postconditions respectively.

2.3 Reversible control flows

One can define reversible `if`, `for` and `while` statements in a reversible program. Fig. 1 (a) shows the flow chart of executing the reversible `if` statement. There are two condition expressions in this chart, a precondition and a postcondition. The precondition decides which branch to enter in the forward execution. After executing the specific branch, the program checks the consistency between precondition and postcondition to make sure they are consistent. To reverse this statement, one can exchange the precondition and postcondition, and reverse the expressions in both branches. Fig. 1 (b) shows the flow chart of the reversible `while` statement. There are also two conditions expressions. Before executing the condition expressions, the program preassumes the postcondition is false. After each iteration, the program asserts the postcondition to be true. To reverse this statement, one can exchange the precondition and postcondition, and reverse the body statements. The reversible `for` statement is similar to the irreversible one except that after execution, the program will assert the iterator to be unchanged. To reverse this statement, one can exchange `start` and `stop` and inverse the sign of `step`.

The following code computes the Fobonacci number recursively and reversibly.

Listing 4: Computing Fibonacci number recursibly and reversibly.

134
```
@i function rrfib(out!, n)
    @invcheckoff if (n >= 1, ~)
        counter ← 0
        counter += n
        while (counter > 1, counter!=n)
            rrfib(out!, counter-1)
            counter -= 2
        end
        counter -= n % 2
        counter → 0
    end
    out! += 1
end
```

Here, `out!` is an integer initialized to `0` for storing outputs. The preconditions and postconditions are wrapped into a tuple. In the `if` statement, the postcondition is the same as the precondition, hence we omit the postcondition by inserting a "~" in the second field as a placeholder. In the while statement, the postcondition is true only for the initial loop. Once code is proven correct, one can turn off the reversibility check by adding `@invcheckoff` before a statement. This will remove the reversibility check and make the code faster and compatible with GPU kernels (kernel functions can not handle exceptions).

2.4 Reverse computing is not checkpointing

There are two approaches to trace back intermediate states in reversible computing, one is checkpointing and another is reverse computing. When we talk about reversible programming languages, we always refers to those implement reverse computing. The physical origin of why people prefer recovering information through reverse computing lies in how reversible computing devices saves energy. Taking adiabatic CMOS as an example, its logic unit can be charged up and charged down (signal recovery). To recover energy in the charging down phase, the output signal must be the same as when it was charged up. This brings challenges to circuit archetecture designs. Naively, one can connect the output of a logic unit to the input of the next one directly in a cascade fashion. While all logic units does not charged down until the computation ends. In cascade stage n, the activity factor scales as $1/n$ and the circuit performance suffers accordingly. Athas and Svensson (1994) Designs considered as more practical, such as SCRL and 2LAL and recent S2LAL use pipeline structure. Where basic instructions are reversible so that signals can be recovered through uncomputing. To compile code to adiabatic CMOS, a programming language must implement reverse computing.

In the most straightforward g-segment tradeoff scheme Bennett (1989); Levine and Sherman (1990); Perumalla (2013), 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 is similar to the checkpointing Griewank and Walther (2008); Chen *et al.* (2016) that widely used in many machine learning frameworks. Both reverse computing and checkpointing can make a program reversible. Checkpointing takes snapshots of the computational state pior to current stage so that all intermediate results can be recomputed. While reverse computing restores the state from the inverse direction and can utilize reversibility to avoid unnessesary allocations. Utilizing reversibility is especially important at the lower level design, which can be seen from the connection between reversibility and adiabatic logic circuit design. Checkpointing shares the same spirit with the cascade layout Hall (1992) for connecting adiabatic logic units. The cascade layout is believe not practical because the very first input (the allocated memory at the checkpoint) must remain valid when the last output is allowed to go invalid. For n level cascading, the activity factor for each stage will descrease as $1/n$, resulting into poor circuit performance. On the other side, the reverse computing correspondence pipeline layout Athas and Svensson (1994) can restore the input signals by running inverse operation of circuit blocks, which has been widely used mordern reversible circuit design Anantharam *et al.* (2004). This is the underlying reason why reversible languages do not use checkpointing. It is a solid proof from the practise that it is the reversible programming rather than the checkpointing that can differentiate the whole language from the machine instruction level. When there is not time overhead, both have a space overhead propotional to time. When a polynomial overhead in time is allowed, reversible computing has a minimum space overhead of $O(S \log(T))$ [Robert Y. Levine, 1990]. While for checkpointing, there can be no space overhead. One can just recompute from beginning to obtain any intermediate state.

Reverse computing shows advantage in handling effective codes with mutable structures and arrays. For example the affine transformation can be implemented without any overhead.

Listing 5: Affine transformation without extra memory cost.

```
@i function i_affine!(y!::AbstractVector{T}, W::AbstractMatrix{T}, b::AbstractVector{T}, x:
    :AbstractVector{T}) where T
    @safe @assert size(W) == (length(y!), length(x)) && length(b) == length(y!)
    @invcheckoff for j=1:size(W, 2)
        for i=1:size(W, 1)
            @inbounds y![i] += W[i,j]*x[j]
        end
    end
    @invcheckoff for i=1:size(W, 1)
        @inbounds y![i] += b[i]
    end
end
```

Reverse computing also provides a path to utilize reversibility. It can differentiate the inplace version of unitary matrix multiplication, which can be used to implement memory efficient recurrent neural networks Jing *et al.* (2016).

Listing 6: Two level decomposition of a unitary matrix.

```
@i function i_umm!(x!::AbstractArray, θ)
    M ← size(x!, 1)
    N ← size(x!, 2)
    k ← 0
    @safe @assert length(θ) == M*(M-1)/2
    for l = 1:N
        for j=1:M
            for i=M-1:-1:j
                INC(k)
                ROT(x![i,l], x![i+1,l], θ[k])
            end
        end
    end
    k → length(θ)
end
```

Last but not least, it encourages users to code in a memory friendly style. Reversible programming does not allocate automatically for people, so that the programmer need to think how to make the program reversible and cache friendly. For example, to compute the power of a positive fixed point number and an integer, one can easily write irreversible code as in Listing. 7

Listing 7: A regular power function.

```
function mypower(x::T, n) where T
    y = one(T)
    for i=1:n
        y *= x
    end
    return y
end
```

Listing 8: A reversible power function.

```
@i function mypower(out, x::T, n) where T
    if (x != 0, ~)
        @routine begin
            ly ← one(ULogarithmic{T})
            lx ← one(ULogarithmic{T})
            lx *= convert(x)
            for i=1:n
                ly *= x
            end
        end
        out += convert(ly)
        ~@routine
    end
end
```

Since fixed point number is not reversible under multiplication, the regular power with checkpointing would require checkpointing inside a loop, which will cause bad performance. With reversible thinking, we can convert the fixed point number to logarithmic numbers for computing as shown in Listing. 8. One can compute the output without sacrificing reversibility. The algorithm to convert a regular fixed point number to a logarithmic number is efficient Turner (2010). Even in cases where allocation inside the loop can not be avoided, reversible programming allows a user to preallocate a chunk outside of the loop, so that computation inside the loop can still be efficient.

3 Reversible automatic differentiation

3.1 First order gradient

If we inline all the instructions, the program would be like Listing. 9. The automatically generated inverse program (i.e. $(y, x) \to (y - \log(x), x))$) is like Listing. 10.

Listing 9: The expanded function body of Listing. **??**.

```
N ← min(length(x), length(y!))
for i=1:N
    @routine begin
        nsq ← zero(T)
        n ← zero(T)
        nsq += x[i].re ^ 2
        nsq += x[i].im ^ 2
        n += sqrt(nsq)
    end
    y![i].re += log(n)
    y![i].im += atan(x[i].im, x[i].re)
    ~@routine
end
N → min(length(x), length(y!))
```

Listing 10: The inverse of Listing. 9.

```
N ← min(length(x), length(y!))
for i=N:-1:1
    @routine begin
        nsq ← zero(T)
        n ← zero(T)
        nsq += x[i].re ^ 2
        nsq += x[i].im ^ 2
        n += sqrt(nsq)
    end
    y![i].re -= log(n)
    y![i].im -= atan(x[i].im, x[i].re)
    ~@routine
end
N → min(length(x), length(y!))
```

To compute the adjoint of the computational process in Listing. 9, one simply insert the gradient code into its inverse in Listing. 10. The resulting code is show in Listing. 11, the original arithmetic instructions are highlighted with yellow background color, they now apply on the value field (`.x`) of the input value. Along with these reversed code, we have inserted a bundle of extra code to update the gradient field (`.g`). `@zeros TYPE var1 var2...` is the macro to allocate multiple ancillas. Since these "allocated" variables are scalars, they do not really access the system memory. Its inverse operations starts with `~@zeros` returns zero emptied ancillas to the system.

Listing 11: Insert the gradient code into Listing. 10.

```
N ← min(length(x), length(y!))
for i=N:-1:1
   @routine begin
       nsq ← zero(GVar{T,T})
       n ← zero(GVar{T,T})

       gsqa ← zero(T)
       gsqa += x[i].re.x * 2
       x[i].re.g -= gsqa * nsq.g
       gsqa -= nsq.x * 2
       gsqa -= x[i].re.x * 2
       gsqa → zero(T)
       nsq.x += x[i].re.x ^2

       gsqb ← zero(T)
       gsqb += x[i].im.x * 2
       x[i].im.g -= gsqb * nsq.g
       gsqb -= x[i].im.x * 2
       gsqb → zero(T)
       nsq.x += x[i].im.x ^2

       @zeros T ra rb
       ra += sqrt(nsq.x)
       rb += 2 * ra
       nsq.g -= n.g / rb
       rb -= 2 * ra
       ra -= sqrt(nsq.x)
       ~@zeros T ra rb
       n.x += sqrt(nsq.x)
   end

   y![i].re.x -= log(n.x)
   n.g += y![i].re.g / n.x

   y![i].im.x-=atan(x[i].im.x,x[i].re.x)
   @zeros T xy2 jac_x jac_y
   xy2 += abs2(x[i].re.x)
   xy2 += abs2(x[i].im.x)
   jac_y += x[i].re.x / xy2
   jac_x += (-x[i].im.x) / xy2
   x[i].im.g += y![i].im.g * jac_y
   x[i].re.g += y![i].im.g * jac_x
   jac_x -= (-x[i].im.x) / xy2
   jac_y -= x[i].re.x / xy2
   xy2 -= abs2(x[i].im.x)
   xy2 -= abs2(x[i].re.x)
   ~@zeros T xy2 jac_x jac_y

   ~@routine
end
```

In really implementation, instead of inserting codes directly, we utilize Julia's multiple dispatch and "insert" the gradient code by overloading the basic instructions for the wrapper type `GVar`. The same strategy has been used in the ForwardDiff package in Julia. Thanks to the just in time compiling technology, the above code does not run as long as it looks. Computing the gradient takes similar time as computing the complex valued log with Julia's builtin log function alone. One does not need to define gradient function for the inversed program in Listing. 10, because taking inverse and computing gradients commute McInerney (2015). Hence, we can simply reverse the gradient function in Listing. 11.

3.2 Hessians

Combining forward mode AD and reverse mode AD 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 examplify it in a benchmark in Sec. 4.2.

3.3 CUDA kernels

CUDA programming is playing a significant role in high-performance computing. In Julia, one can write GPU compatible functions in native Julia language with KernelAbstractions Besard *et al.* (2017). Since NiLang does not push variables into stack automatically for users, it is safe to write differentiable GPU kernels with NiLang. We will show this feature in the benchmarks of bundle adjustment (BA) in Sec. 4.3. Here, one should notice that the shared read in forward pass will become shared write in the backward pass, which may result in incorrect gradients. We will review this issue in the supplimentary material.

4 Benchmarks

It is interesting to see how does our framework comparing with the state-of-the-art GP-AD frameworks, including source code transformation based Tapenade and Zygote and operator overloading based ForwardDiff and ReverseDiff. Since most DS-AD packages like famous Tensorflow and Pytorch are not dessigned for the using cases used in our benchmarks, we do not include those package to avoid an unfair comparison. 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 the reversibility check off to achieve a better performance. Codes used in benchmarks could be found the in Examples section of the supplimentary material.

4.1 Sparse matrices

We benchmarked the call, uncall and backward propagation 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, since `mul!` 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 1: 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 by summing “O” and “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 arithmetic instructions is longer than pure computing.

4.2 Graph embedding problem

Since one can combine ForwardDiff and NiLang to obtain Hessians, it is interesting to see how much performance we can get in differentiating the graph embedding program. The problem definition could be found in the supplimentary material.

In Table 2, we show the the performance of different implementations by varying the dimension k. The number of parameters is $10k$. As the baseline, (a) shows the time for computing 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 due to the uncomputing overhead. 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, where the combo of NiLang and ForwardDiff gives the best performance for $k \geq 3$.

k	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
ReverseDiff-G	2.823e-05	4.582e-05	6.045e-05	7.651e-05	9.666e-05
ForwardDiff-G	1.518e-05	4.053e-05	6.732e-05	1.184e-04	1.701e-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 2: Absolute times in seconds for computing the objectives (O), uncall objective (U), gradients (G) and Hessians (H) of the graph embedding program. k is the embedding dimension, the number of parameters is $10k$.

4.3 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-writing the programs in a reversible style. We show the results in Table 3 and Table 4. In our new benchmarks, we also 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 provides a baseline for comparison.

# 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	9.844e-03	1.166e-02	2.797e-01	9.745e-02	3.903e-02	7.476e-02	2.284e-01	3.593e+00
NiLang-O	3.655e-03	1.425e-02	1.040e-01	1.389e-01	7.388e-02	1.491e-01	4.176e-01	5.462e+00
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	3.551e-02	1.673e+00	4.811e+01	1.599e+02	-	-	-	-
NiLang-G	9.102e-03	3.709e-02	2.830e-01	3.556e-01	6.652e-01	1.449e+00	3.590e+00	3.342e+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 3: Absolute runtimes in seconds for computing the objective (O) and gradients (G) of GMM with 10k data points. "-" represents missing data due to not finishing the computing in limited time.

In the GMM benchmark, NiLang's objective function has overhead comparing with irreversible programs in most cases. Except the uncomputing overhead, it is also because our naive reversible matrix-vector multiplication is much slower than the highly optimized BLAS function, where the matrix-vector multiplication is the bottleneck of the computation. The forward mode AD suffers from too large input dimension in the large number of parameters regime. Although ForwardDiff batches the gradient fields, the overhead proportional to input size still dominates. The source to source AD framework Tapenade is faster than NiLang in all scales of input parameters, but the ratio between computing the gradients and the objective function are close.

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

In the BA benchmark, reverse mode AD shows slight advantage over ForwardDiff. 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 the output space. In this instance, our reversible implementation is even faster than the source code transformation based AD framework Tapenade. Comparing with Tapenade that inserting stack operations into the code automatically. NiLang gives users the flexibility to memory management, so that the code can be compiled to GPU. With KernelAbstractions, we compile our reversible program to GPU with no more than 10 lines of code, which provides a >200x speed up.

5 Nitpicking NiLang

Although there is no limitation in writing a general program in a reversible form. It is generally hard for one to get used to this programming style. It is a chanllenge for authors of this paper to figure out the design patterns in reversible programming too. With more and more experience, we find writting a reversible program is just as simple as writting a regular program.

> The strangeness of the reversible programming style is due mainly to our lack of experience with it. – Baker (1992)

The main limitation of NiLang is using floating point number might cause the accumulation of rounding errors. A better number system for a reversible programming language might be a combination of fixed point numbers and logarithmic numbers. Most analytic functions can be computed by Taylor explasion with constant memory and time overhead. One can see supplimentary material for an example of computing the Bessel function.

Broader Impact

Our automatic differentiation in a reversible eDSL brings the field of reversible computing to the modern context. We believe it will be accepted by the public to meet current scientific automatic differentiation needs and aim for future energy-efficient reversible devices. For solving practical issues, in an unpublished paper, we have successfully differentiated a spin-glass solver to find the optimal configuration on a 28×28 square lattice in a reasonable time. There are also some interesting applications like normalizing flow and bundle adjustment in the example folder of NiLang repository and JuliaReverse organization. For the future, energy consumption is an even more fundamental issue than computing time and memory. Current computing devices, including CPU, GPU, TPU, and NPU consume much energy, which will finally hit the "energy wall". We must get prepared for the technical evolution of reversible computing (quantum or classical), which may cost several orders less energy than current devices.

We also see some drawbacks to the current design. It requires the programmer to change to programing style rather than put effort into optimizing regular codes. It is not fully compatible with modern software stacks. Everything, including instruction sets and BLAS functions, should be redesigned to support reversible programming better. We put more potential issues and opportunities in the discussion section of the supplementary material. Solving these issues requires the participation of people from multiple fields.

References

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.

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] .

W. Zhu, K. Xu, E. Darve, and G. C. Beroza, "A general approach to seismic inversion with automatic differentiation," (2020), arXiv:2003.06027 [physics.comp-ph] .

V. Bergholm, J. Izaac, M. Schuld, C. Gogolin, M. S. Alam, S. Ahmed, J. M. Arrazola, C. Blank, A. Delgado, S. Jahangiri, K. McKiernan, J. J. Meyer, Z. Niu, A. Száva, and N. Killoran, "Pennylane: Automatic differentiation of hybrid quantum-classical computations," (2018), arXiv:1811.04968 [quant-ph] .

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

C. Roberts, A. Milsted, M. Ganahl, A. Zalcman, B. Fontaine, Y. Zou, J. Hidary, G. Vidal, and S. Leichenauer, "Tensornetwork: A library for physics and machine learning," (2019), arXiv:1905.01330 [physics.comp-ph] .

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).

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] .

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

J. Utke, U. Naumann, M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill, and C. Wunsch, ACM Trans. Math. Softw. **34** (2008), 10.1145/1377596.1377598.

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 .

Y. Shen and Y. Dai, IEEE Access **6**, 11146 (2018).

G. Forget, J.-M. Campin, P. Heimbach, C. N. Hill, R. M. Ponte, and C. Wunsch, (2015).

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

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

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.

L. Dinh, D. Krueger, and Y. Bengio, "Nice: Non-linear independent components estimation," (2014), arXiv:1410.8516 [cs.LG] .

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 .

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).

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).

D. Speyer and B. Sturmfels, Mathematics Magazine **82**, 163 (2009).

W. C. Athas and L. Svensson, in *Proceedings Workshop on Physics and Computation. PhysComp'94* (IEEE, 1994) pp. 111–118.

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

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

A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation* (SIAM, 2008).

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

J. S. Hall, in *Proceedings of Physics of Computation Workshop, Dallas Texas* (Citeseer, 1992).

V. Anantharam, M. He, K. Natarajan, H. Xie, and M. P. Frank, in *ESA/VLSI* (2004) pp. 5–11.

L. Jing, Y. Shen, T. Dub**c**ek, J. Peurifoy, S. Skirlo, Y. LeCun, M. Tegmark, and M. Solja**c**ić, "Tunable efficient unitary neural networks (eunn) and their application to rnns," (2016), arXiv:1612.05231 [cs.LG] .

C. S. Turner, IEEE Signal Processing Magazine **27**, 124 (2010).

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

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

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

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

H. G. Baker, in *International Workshop on Memory Management* (Springer, 1992) pp. 507–524.