Differentiate Everything with a Reversible Embeded Domain-Specific Language

Anonymous authors
Paper under double-blind review

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) softwares 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, they use these tools 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 sometimes has to defined backward rules manually, for example

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

To avoid defining backward rules manually, one can also use a general purposed AD (GP-AD) software 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). However, these softwares 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. Although most of them are source code transformation based, they can not be used to differentiate GPU kernel functions because implicit stack operations are not compatible with kernel functions.

We need a new 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), where intermediate states can be traced backward without accessing an implicit stack. Reversible programming provides flexible time-space tradeoff. It also allows people to utilize the reversibility to reverse a program without any overhead. In machine learning, reversibility 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). Reversible programming language will makes these happen naturally.

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) such as 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); Takeuchi *et al.* (2014; 2017), and these reversible computing device can be orders more energy-efficient. Landauer Landauer (1961) proves that only when a device does not erase information (i.e. reversible), its energy efficient can go beyond certern thermal dynamic limit. However, these reversible programming languages can not be used in real scientific computing, 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) as a new playground of GP-AD.

In this paper, we first introduce the language design of NiLang in Sec. 2. In Sec. 3, we explain the implementation of automatic differentiation in NiLang. In Sec. 4, we benchmark the performance of NiLang's AD with other AD softwares and explain why it is fast.

2 Language design

NiLang is 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. Meanwhile, it has rich features for meta-programming. Its package for pattern matching MLStyle allows us to define an eDSL 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. It also implements the compute-copy-uncompute Bennett (1973) macro to increase the code reusability. 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. NiLang's source code is available online https://github.com/GiggleLiu/NiLang.jl, https://github.com/GiggleLiu/NiLangCore.jl. By the time of writting, the version of NiLang is v0.7.2.

2.1 Reversible functions and arithmetic instructions

Mathematically, any irreversilbe mapping `y = f(args...)` can be trivially transfromed to its reversible form `y += f(args...)` or `y ⊻= f(args...)` (`⊻` is the bitwise `XOR`), where `y` is a pre-emptied variable. But in numeric computing with finite precision, this is not always true. 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. Other reversible operations includes `SWAP`, `ROT`, `NEG` et. al., and this instruction set is extensible.

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 $\mathbb{R}^3 \rightarrow \mathbb{R}^3$. The ! after a symbol is a part of the name, as a convension to indicate the mutated variables.

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 pre-emptied memory as `x ← 0`, and its inverse deallocating a **zero emptied** variable as `x → 0`. An unknown variable can be pushed to a stack and used in the uncomputing stage with a pop statement. If a variable is allocated and deallocated in the local scope, we call it an ancilla.

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

```
@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 inverse of the top two lines. i.e., the bottom two lines *uncomputes* the top two lines. The motivation of uncomputing 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.

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)

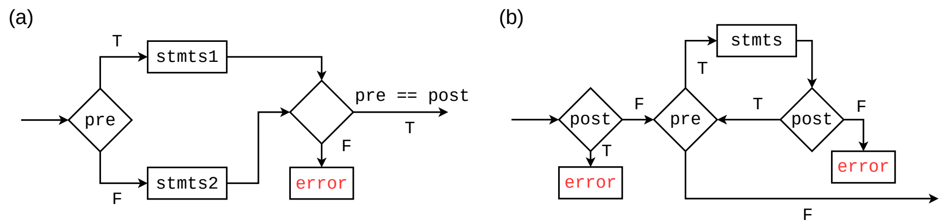

Figure 1: The flow chart for reversible (a) `if` statement and (b) `while` statement. "pre" and "post" represents precondition and postconditions respectively.

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.

```
@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 of a computational process, one is checkpointing Griewank and Walther (2008); Chen *et al.* (2016) and another is reverse computing. Reverse computing and checkpointing share many similarities. When there is no time overhead, both have a space overhead $\sim O(T)$ (i.e. linear to time). They also have many differences. When a polynomial overhead in time is allowed, reversible computing has a minimum space overhead of $O(S\log(T))$ Bennett (1989); Levine and Sherman (1990); Perumalla (2013). For checkpointing, there can be no space overhead, since one can just recompute from beginning to obtain any intermediate state with a time complexity $O(T^2)$. In practical using cases, reverse computing has the following advantages.

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: Inplace affine transformation.

```
@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
```

Here, the expression following the `@safe` macro is an external irreversible statement.

Reverse computing can utilize reversibility to trace back states without extra memory cost. For example, we can define the unitary matrix multiplication that can be used in a type of memory efficient recurrent neural network 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, reversible programming encourages users to code in a memory friendly style. Since allocations in reversible programming are explicit, programmers have the flexibility to control how to allocate memory and which number system to use. 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::Int) 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::Int) 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 *=, naive checkpointing would require stack operations inside a loop. With reversible thinking, we can convert the fixed point number to logarithmic numbers to utilize the reversibility of *= as shown in Listing. 8. Here, the algorithm to convert a regular fixed point number to a logarithmic number can be efficient Turner (2010).

3 Reversible automatic differentiation

3.1 Back propagation

To back propagate the program, we first reverse the code through source code transformation, and then insert the gradient code through operator overloading. If we inline all the instructions in Listing. 3, 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 function body of Listing. 3.

```
@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
```

Listing 10: The inverse of Listing. 9.

```
@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
```

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 inlined code is show in Listing. 11.

Listing 11: Insert the gradient code into Listing. 10, original computational processes are highlighted in yellow background.

```
@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
   rta += 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
```

Here, `@zeros TYPE var1 var2...` is the macro to allocate multiple variables of the same type. Its inverse operations starts with `~@zeros` deallocates zero emptied variables. In practise, "inserting gradients" is not achieved by source code transformation, but by changing the element type to `GVar`, a composite type with two fields, value `x` and gradient `g`. With multiple dispatching primitive instructions on this new type, values and gradients can be updated simultaneously. Although the code looks much longer, the computing time (with reversibility check closed) is not.

Listing 12: Time and allocation to differentiate complex valued log.

```
julia> @inline function (ir_log)(x::Complex{T}) where T
           log(abs(x)) + im*angle(x)
       end

julia> @btime ir_log(x) setup=(x=1.0+1.2im); # native code
  30.097 ns (0 allocations: 0 bytes)

julia> @btime (@instr y += log(x)) setup=(x=1.0+1.2im; y=0.0+0.0im); # reversible code
  17.542 ns (0 allocations: 0 bytes)

julia> @btime (@instr ~(y += log(x))) setup=(x=GVar(1.0+1.2im, 0.0+0.0im); y=GVar(0.1+0.2im
     , 1.0+0.0im)); # adjoint code
  25.932 ns (0 allocations: 0 bytes)
```

The performance is unreasonably good because the generated Julia code is further compiled to LLVM so that it can enjoy existing optimization passes. For example, the optimization passes can find out that for a irreversible device, uncomputing local variables `n` and `nsq` has no effect the to return values, so that the code uncomputing will be ignored automatically. Unlike checkpointing based approaches that focus a lot in the optimization of data caching on a global stack, NiLang does not have any optimization pass in itself. Instead, it throws itself to existing optimization passes in Julia. Without accessing global stack, NiLang's code is quite friendly to optimization passes. In this case, we also see the boundary between source code transformation and operator overloading can be vague in a Julia, in that the generated code can be very different from how it looks.

One can define the adjoint of a primitive instruction as a reversible function on **either** the function itself or its inverse, because the adjoint program of reversible functions are reversible to each other too.

$$f : (\vec{x}, \vec{g}_x) \rightarrow (\vec{y}, \frac{\partial \vec{y}}{\partial \vec{x}} \vec{g}_x) \tag{1}$$

$$f^{-1} : (\vec{y}, \vec{g}_y) \rightarrow (\vec{x}, \frac{\partial \vec{x}}{\partial \vec{y}} \vec{g}_y) \tag{2}$$

It can be easily verified by applying the above two mappings concequtively, which turns out to be an identity mapping considering $\frac{\partial \vec{y}}{\partial \vec{x}} \frac{\partial \vec{x}}{\partial \vec{y}} = \mathbb{1}$. As an example, the joint functions for primitive instructions `(:+=)(sqrt)` and `(:-=)(sqrt)` used above can be defined as in Listing. 13.

Listing 13: Adjoints for primitives `(:+=)(sqrt)` and `(:-=)(sqrt)`.

```
@i @inline function (:-=)(sqrt)(out!::GVar, x::GVar{T}) where T
    @routine @invcheckoff begin
        @zeros T a b
        a += sqrt(x.x)
        b += 2 * a
    end
    out!.x -= a
    x.g += out!.g / b
    ~@routine
end
```

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. We will use this approach to compute Hessians in the graph embeding benchmark in Sec. A.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 differentiate CUDA kernels with no more than extra 10 lines in the bundle adjustment (BA) benchmark in Sec. 4.1.

4 BENCHMARKS

We benchmark our framework 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 tensor based AD softwares 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 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 1 and Table 2. 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 1: 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.

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.

You can find more benchmarks in Appendix A, including differentiating sparse matrix dot product and obtaining Hessians in the graph embedding application.

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

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

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).

N. Takeuchi, Y. Yamanashi, and N. Yoshikawa, Scientific reports **4**, 6354 (2014).

N. Takeuchi, Y. Yamanashi, and N. Yoshikawa, Scientific reports **7**, 1 (2017).

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).

C. H. Bennett (1973).

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

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 .

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

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

L. Jing, Y. Shen, T. Dubcek, J. Peurifoy, S. Skirlo, Y. LeCun, M. Tegmark, and M. Soljacić, "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).

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).

J.-G. Liu, L. Wang, and P. Zhang, "Tropical tensor network for ground states of spin glasses," (2020), arXiv:2008.06888 [cond-mat.stat-mech] .

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

A More Benchmarks

A.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 3: 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.

A.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.

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 4: 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 $10k$.

In Table 4, 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 $k \geq 3$.

Discussions

Our automatic differentiation in a reversible eDSL brings the field of reversible computing to the modern context. Reversible computing has been gradually accepted as the only approach to break the curse of energy-efficiency. However, it is completely not compatible with current software stack.

We verified that reversible software stack can be a power tool on its own in the domain of scientific AD. In fact, we have already used it to differentiated a spin-glass solver to find the optimal configuration on a 28×28 square lattice in a reasonable time Liu *et al.* (2020), which is not possible without flexible memory management. 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.

Although reversible programming is equally powerful as regular programming. It is generally hard for one to get used to this programming style. 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)

A.3 NITPICKING NILANG

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. In the previous text, we use NiLang to differentiate CUDA kernel functions, one should notice that the shared read in the forward pass will become shared write in the adjoint pass when updating the gradient fields, which may result in incorrect gradients.