Differentiate Everything with a Reversible Domain-Specific Language

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

Traditional machine reverse-mode automatic differentiation (AD) suffers from the problem of having a space overhead that linear to time in order to trace back the computational state, which is also the source of poor performance. In reversible programming, a program can be executed bi-directionally, which means we do not need any additional design to trace back the computational state. This paper answers how practical it is to implement a programming language level reverse mode AD in a reversible programming language. By implementing sparse matrix operations and some machine learning applications in our reversible eDSL NiLang, and benchmark the performance with state-of-the-art AD frameworks, our answer is a definite positive. NiLang is an open-source r-Turing complete reversible eDSL in Julia. It empowers users the flexibility to tradeoff time, space, and energy rather than caching data into a global tape. Manageable memory allocation makes it an excellent tool to differentiate GPU kernels too.

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. These frameworks sometimes fail to meet the diverse needs in research, for example, in physics research,

1. People 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. People 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); Seeger *et al.* (2017); Wan and Zhang (2019); Hubig (2019),
3. People 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).

People have to keep adding new backward rules to the function pool. In the remaining text, we call this type of AD the domain specific AD (DS-AD). To meet the diversed need, we need a general purposed AD (GP-AD) too that differentiate a general program, including scalar operations efficiently. Some source code transformation based AD packages like Tapenade Hascoet and Pascual (2013) and Zygote Innes (2018); Innes *et al.* (2019) are close to this goal. They read the source code from a user and generate a new code that computes the gradients. However, these

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

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, and the memory usage will become a bottleneck as well. Caching automatically for users also makes the code not compatible to GPU, it is a huge loss for the a language that supporting compiling generic codes to GPU devices like Julia Bezanson *et al.* (2012, 2017).

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. 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). These reversible languages have solid design of reversible memory management so that the memory allocation, or the time-space tradeoff is well under the programmers' control. A reversible language has a natural trait that they can make use of reversibility so that there is no extra time or space cost to trace back a reversible operation. In machine learning, people also manage to not erasing informations that needed in the backward propagation. These neural networks includes 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) with reversible activation functions. Utilizing reversibility is proven to decrease the memory usage by two orders in some cases, most of these applications can be written in a reversible programming language naturally without extra framework designs. Reversible programming can generalize this idea to elementary scalar operations so that programmers' reversible thinking can help make use the reversibility more extensively to differentiate the whole programming lanauge.

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 from the perspective of information and entropy, or by the Landauer's principle Landauer (1961). People tries to keep the language restrictive and well defined so that they can be compiled to future hardwares. The drawback is they can be hardly used in real computation directly, most of them do not have basic elements like floating point numbers, arrays and complex numbers that are useful in scientific computing. Not to say most of them do not have a well maintained compiler to help simulate the code on a regular device. This motivates us to build a new embeded domain specific language (eDSL) in Julia to solve these issues, so that it can be used directly to accelerate machine learning frameworks in the host language.

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

2.1 A general introduction to the reversible language design

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 `x ← 0`, and the corresponding deallocation as `x → 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

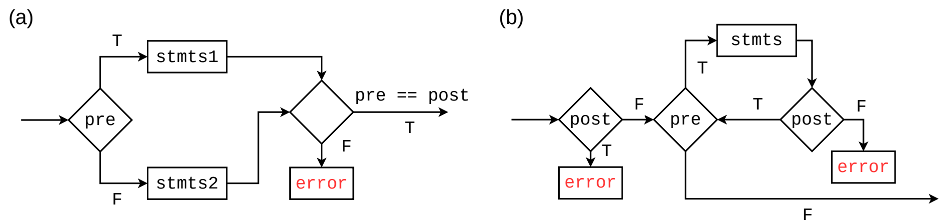

Figure 1: The flow chart for reversible (a) `if` statement and (b) `while` statement. "pre" and "post" represents precondition and postconditions respectively.

memory when deallocating is not easy. Hence Charles H. Bennett introduced the famous compute-copy-uncompute paradigm Bennett (1973) for reversible programming.

2.1.2 Control flows

One can define reversible `if`, `for` and `while` statements in a slightly different way comparing with its irreversible counterpart. The reversible `if` statement is shown in Fig. 1 (a). Its condition statement contains two parts, 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 in Fig. 1 (b) also has two condition fields. 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, `y ⊻= f(args...)` is self reversible.
- For integer and floating point arithmetic operations, we treat `y += f(args...)` and `y -= f(args...)` as reversible to each other. Here `f` can be an arbitrary pure function such as `identity`, `*`, `/` and `^`. Let's forget the floating point rounding errors for the moment and discuss in detail in the supplimentary materials.
- For logartihmic number and tropical number algebra Speyer and Sturmfels (2009), `y *= f(args...)` and `y /= f(args...)` as reversible to each other. Notice the zero element ($-\infty$) in the Tropical algebra is not considered here.

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.

Although there are a lot reversible programming language candidates, they lack the basic components for scientific programming like arrays and complex numbers, and most of them are designed as a stand alone language that can not be embeded in other machine learning frameworks. Hence we develop an embedded domain-specific language (eDSL) NiLang in Julia language Bezanson *et al.* (2012, 2017) that implements reversible programming. One can write reversible control flows, instructions, and memory managements inside a macro. Julia is a popular language for scientific programming. We choose Julia as the host language for multiple purposes. The most important consideration is speed that crucial for a GP-AD. Its clever design of type inference and just in time compiling provides a C like speed. Also, it has a rich ecosystem for meta-programming. The package for pattern matching MLStyle allow us to define an eDSL conveniently. Last but not least, its multiple-dispatch provides the polymorphism that will be used in our AD engine. Comparing with a regular reversible programming language, NiLang features containing many practical elements for scientific computing like array operations and rich number systems, including floating point number, complex number, fixed point number and logarithmic number. It also assumes the floating point += and −= operations are reversible to each other and introduces the concept of *dataview*, the bijective mapping of a content in the memory, to allow flexible data field access to structural instances in the host language, All these changes are motivated by making it a practical platform for differential applications, even though including floating point numbers might be incompatible with reversible hardwares. By the time of writting, the version of NiLang is v0.7.2. Let's start by defining a reversible adder.

Listing 1: A reversible adder

```julia
@i function adder(y!::Real, x::Real)
    y! += x
end

@assert adder(2, 3) == (5, 3)
@assert (~adder)(5, 3) == (2, 3)
```

Macro `@i` generates two functions that reversible to each other `adder` and `~adder`, each defines a mapping $\mathbb{R}^2 \to \mathbb{R}^2$. The ! after a symbol is a part of the name to indicate that a variable is changed. A reversible += instruction is always defined as `y += f(args...)`, where `f` is a mapping that allows to be irreversible, or just leave empty for identity mapping. We can easily check these two functions are reversible to each other. Then let's see a more complicated example of computing the complex valued log (a built in function).

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

```julia
@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
```

Here, the macro `@inline` tells the compiler that this function can be inlined. `n ← zero(T)` is the ancilla allocation statement. One can input "←" and "→" by typing "\leftarrow[TAB KEY]" and "\rightarrow[TAB KEY]" respectively in a Julia editor or REPL. `@routine` and `~@routine` are macros for computing and uncomputing. i.e. `~@routine` means running the statement marked with `@routine` backwards. One can use the `begin ... end` statement to wrap multiple statements as one. NiLang view every field of a variable as mutable, so that the real part (`y!.re`) and imaginary (`y!.im`) of a complex number can also be changed directly.

We can want to apply this `log` function to an array, we can define

Listing 3: Applying the log function to an array.

```julia
@i function broadcasted_log!(y!::Array{Complex{T}, N}, x::Array{Complex{T}, N}) where T
    N ← min(length(x), length(y!))
    for i=1:N
        y![i] += log(x[i])
    end
end
```

2.2 Reverse computing is not checkpointing

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); 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

2.2.1 handling mutable arrays

2.2.2 utilizing reversibility

2.2.3 helping users code better

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. 4

Listing 4: 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 5: 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. 5. 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. 6. The automatically generated inverse program (i.e. $(y, x) \rightarrow (y - \log(x), x))$) is like Listing. 7.

Listing 6: The expanded function body of Listing. 3.

```
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 7: The inverse of Listing. 6.

```
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. 6, one simply insert the gradient code into its inverse in Listing. 7. The resulting code is show in Listing. 8, 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 8: Insert the gradient code into Listing. 7.

```
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. 7, because taking inverse and computing gradients commute McInerney (2015). Hence, we can simply reverse the gradient function in Listing. 8.

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.

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 $10k$.

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 $k \geq 3$.

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.

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

# 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.

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] .

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.

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] .

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] .

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

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 .

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 (2017).

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. 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 .

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).

C. H. Bennett (1973).

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

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).

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

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

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.