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 back-propagation. 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 our framework has the state-of-the-art performance.

1 Introduction

Most of the popular automatic differentiation (AD) tools 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 in machine learning. Later, People in the scientific computing domain also realized the power 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); Luo et al. (2019) and variational tensor network simulation Liao et al. (2019); Roberts et al. (2019). To meet the diverse need in these applications, one sometimes has to define backward rules manually, for example

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

Instead of defining backward rules manually, one can also use a general purposed AD (GP-AD) framework like Tapenade Hascoet & Pascual (2013), OpenAD Utke et al. (2008) and Zygote Innes (2018); Innes et al. (2019). Researchers have used these tools in practical applications such as bundle adjustment Shen & Dai (2018) and earth system simulation Forget et al. (2015), where differentiating scalar operations is important. However, the power of these tools are often limited by their relatively poor performance. In many practical applications, a program might do billions of computations. In each computational step, the AD engine might cache some data for backpropagation Griewank & Walther (2008). Frequent caching of data slows down the program significantly, while the memory usage will become a bottleneck as well. Caching implicitly also make these frameworks incompatible with kernel functions. To avoid such issues, we need a new GP-AD framework that does not cache automatically for users.

In this paper, 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 allows people to utilize the reversibility to reverse a program. In machine learning, reversibility is proven to substantially decrease the memory usage in unitary recurrent neural networks MacKay et al. (2018), normalizing flow Dinh et al. (2014), hyper-parameter learning Maclaurin et al. (2015) and residual neural networks Gomez et al. (2017); Behrmann et al. (2018). Reversible programming will make these happen naturally. The power of reversible programming is not limited to handling these reversible applications, any program can be written in a reversible style. Converting an irreversible program to the reversible form would cost overheads in time and space. Reversible programming provides a flexible time-space trade-off scheme that different with checkpointing Griewank (1992), *reverse computing* Bennett (1989); Levine & Sherman (1990), to let user handle these overheads explicitly.

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 & Knight Jr (1999) such as adiabatic complementary metal-oxide-semiconductor (CMOS) Koller & 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 devices are orders more energy-efficient. Landauer Landauer (1961) proves that only when a device does not erase information (i.e. reversible), its energy efficiency can go beyond the thermal dynamic limit. However, these reversible programming languages can not be used directly 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 embedded 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. 3. In Sec. 4, we explain the implementation of automatic differentiation in NiLang. In Sec. 5, we benchmark the performance of NiLang's AD with other AD software and explain why it is fast.

2 Reverse computing style memory management

There are two approaches to trace back intermediate states of a computational process, one is checkpointing Griewank & 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 $O(S\log(T/S))$ Bennett (1989); Levine & 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 time complexity $O(T^2)$. In practical using cases, we need to trade off space and time. The most successful checkpointing algorithm that widely used in automatic differentiation is the treeverse algorithm in Fig. 1(a). Where the computational process is binomially partitioned into sectors. At the begining of each sector, a snapshot is stored into the main memory. The states in the last sector is retrieved by above space efficient $O(T^2)$ algorithm. After that, the last checkpoint can be freed. The remaining sectors are further partitioned into d sub-sectors. The earlier sectors has more quota of snapshots while the later has less, so that the total number of snapshots are remained the same. Recursively apply this treeverse algorithm, the time and space overheads are both logarithmic. The approximated overhead in time and space are

$$T' = tT \tag{1}$$
$$S' = dS \tag{2}$$

Where $T = \eta(t, d)$. One can easily find a solution that both t and d are logarithmic functions of time.

On the other side, the optimal tradeoff of reverse computing also has a recursive structure as shown in Fig. 1 (b). The program is partitioned evenly into k sectors, in each sector, the program is computed in a reversible compute-copy-uncompute style. Both compute and uncomputed are reversible

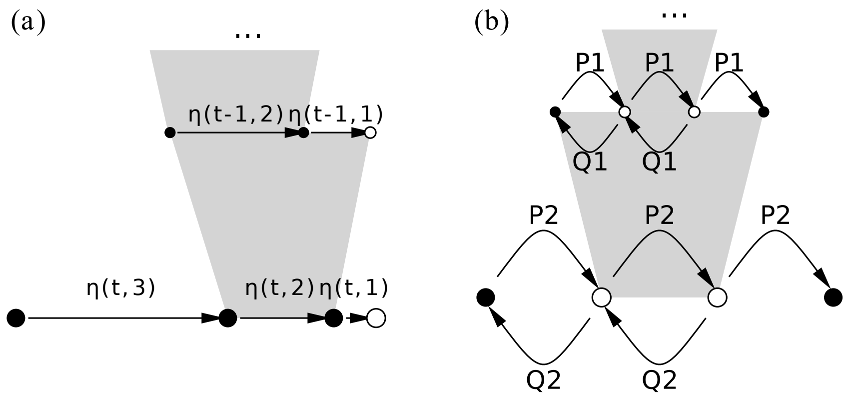

Figure 1: (a) Treeverse algorithm for optimal checkpointing Griewank (1992). $\eta(n,k) \equiv \binom{n+k}{k}$ is the binomial function. (b) Bennett's time space tradeoff scheme for reverse computing Bennett (1973); Levine & Sherman (1990) P and Q are computing and uncomputing respectively.

programs that can be further devided in to k sub-sectors.

$$T' = T\left(\frac{T}{S}\right)^{\frac{\ln(2-(1/k))}{\ln k}} \tag{3}$$

$$S' = \frac{k-1}{\ln k} S \log \frac{T}{S} \tag{4}$$

Here, the overhead in time is polynomial. The treeverse like partition does not apply here because one can not complete the first sweep to create initial checkpoints without introducing any space overheads.

Since the 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. 18. Here, the algorithm to convert a regular fixed-point number to a logarithmic number can be efficient Turner (2010).

Let's start from the worse case that none of the operation in the computation is `intrinsically` reversible. The word intrinsics means not allocating additional memory comparing with its irreversible counter part or mathematical representation.

Consider a program $f_n \circ f_{n-1} \circ ... \circ f_1$ that none of f_i is bijective. The time-efficient reversible algorithm can be defined as in Listing. 3.

Listing 1: Reverse computing space efficiently

```
compute r_{n-1}

# copy results to a pre-emptied memory
r_n += f_n(r_{n-1})

uncompute r_{n-1}
```

Listing 2: Checkpointing space efficiently

```
# forward
pass

# backward
r_n = f_n ∘ f_{n-1} ∘ ... ∘ f_1(x)
r_{n-1} = f_{n-1} ∘ f_{n-2} ... ∘ f_1(x)
...
r_1 = f_1(x)
```

Listing 3: Reverse computing time efficiently

```
# forward, input (x, r_n = 0, STACK)
r_1 ← 0
r_1 += f_1(x)
x → STACK
r_2 ← 0
r_2 += f_2(r_1)
r_1 → STACK
...
r_n ← 0
r_n += f_n(r_{n-1})
r_{n-1} → STACK

# bakcward, input (x, r_n, STACK)
r_{n-1} ← STACK
r_n -= f_n(r_{n-1})
r_1 → 0
...
r_1 ← STACK
r_2 -= f_2(r_1)
r_1 → 0
x ← STACK
r_1 -= f_1(x)
r_1 → 0
```

Listing 4: Checkpointing time efficiently

```
# forward, input (x, STACK)
r_1 = f_1(x)
r_1 → STACK
r_2 = f_2(r_1)
r_2 → STACK
...
r_n = f_n(r_{n-1})
r_n → STACK

# backward, input (STACK)
r_n ← STACK
r_{n-1} ← STACK
...
r_1 ← STACK
```

NOTE: the push operation in reversible context zero-empties the variable automatically. The push in the checkpointing scheme deallocate the variable directly.

NOTE2: Morden compiler helps!

Listing 5: Reverse computing with reversibility

```
# forward, input (x)
f_1(x)
f_2(x)
...
f_n(x)

# bakcward, input (x)
f_n^{-1}(x)
...
f_2^{-1}(x)
f_1^{-1}(x)
```

Method	most time efficient (Time/Space)	most space efficient (Time/Space)	intermediate
Checkpointing	$(T)/(T+S)$	$(T^2)/(S)$	$\sim \frac{t^4}{24}/dS$
Reverse computing (irreversible limit)	$(T)/(T+S)$	$(T(\frac{T}{S})^{0.585})/(S\log(\frac{T}{S}))$	$T(\frac{T}{S})^{\epsilon(k)}/c(k)S\log(\frac{T}{S})$
Reverse computing (reversible limit)	$(T)/(S)$	$(T)/(S)$	T/S

Table 1: T and S are the time and space of the original program. In the reverse computing algorithm, $\epsilon(k) = \frac{\ln(2-(1/k))}{\ln k}$ and $c(k) = \frac{k-1}{\ln k}$. In the treeverse algorithm, the t and d satisfies $T = C_{t+d}^{d}$

The same amount of memory is required by the checkpointing method. These two examples are used to demonstrate the two limits, in practise, we need to balance time and space.

3 Language design

NiLang is an embedded domain-specific language (eDSL) NiLang built on top of the host 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 numbers, complex numbers, fixed-point numbers, and logarithmic numbers. It also implements the compute-copy-uncompute Bennett (1973) macro to increase 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 users to extend instruction sets and sometimes inserting external statements. These features are not compatible with future reversible hardware. NiLang's source code is available online https://github.com/GiggleLiu/NiLang.jl, https://github.com/GiggleLiu/NiLangCore.jl. By the time of writing, the version of NiLang is v0.7.3.

3.1 Reversible functions and instructions

Mathematically, any irreversilbe mapping `y = f(args...)` can be trivially transformed to its reversible form `y += f(args...)` or `y ⊻= f(args...)` ($\veebar$ is the bit-wise `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 & 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. One can define a reversible multiplier in NiLang as in Listing. 6.

Listing 6: A reversible multiplier

```
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 conversion to indicate the mutated variables.

3.2 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 not be dealocate, but can be pushed to a stack pop out later in the uncomputing stage. If a variable is allocated and deallocated in the local scope, we call it an ancilla. Listing. 7 defines the complex valued accumulative log function.

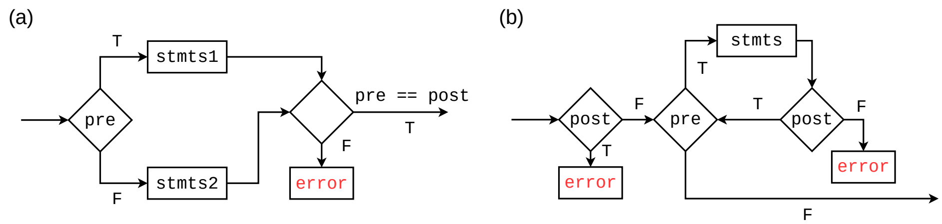

Figure 2: The flow chart for reversible (a) `if` statement and (b) `while` statement. "pre" and "post" represents precondition and postcondition respectively.

Listing 7: Reversible complex valued log function y += $\log(|x|) + i\text{Arg}(x)$.

```
@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 8: Compute-copy-uncompute version of Listing. 7

```
@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. 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. 8.

3.3 Reversible control flows

One can define reversible `if`, `for` and `while` statements in a reversible program. Fig. 2 (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. 2 (b) shows the flow chart of the reversible `while` statement. It also has two condition expressions. Before executing the condition expressions, the program presumes 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`. Listing. 9 computes the Fibonacci number recursively and reversibly.

Listing 9: Computing Fibonacci number recursively 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 precondition and postcondition 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).

4 Reversible automatic differentiation

4.1 Backpropagation

To backpropagate 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 functions in Listing. 8, the function body would be like Listing. 10. The automatically generated inverse program (i.e. $(y, x) \rightarrow (y - \log(x), x)$) would be like Listing. 11.

Listing 10: The inlined function body of Listing. 8.

```
@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 11: The inverse of Listing. 10.

```
@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. 10, one simply insert the gradient code into its inverse in Listing. 11. The resulting inlined code is show in Listing. 12.

Listing 12: Insert the gradient code into Listing. 11, the 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 practice, "inserting gradients" is not achieved by source code transformation, but by operator overloading. We change the element type to a composite type `GVar` 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 13: 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 an irreversible device, uncomputing local variables `n` and `nsq` to zeros does not affect return values, so that it will ignore the uncomputing process 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 the 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 adjoints of reversible functions are reversible to each other too.

$$f : (\vec{x}, \vec{g}_x) \to (\vec{y}, \frac{\partial \vec{y}}{\partial \vec{x}} \vec{g}_x) \tag{5}$$

$$f^{-1} : (\vec{y}, \vec{g}_y) \to (\vec{x}, \frac{\partial \vec{x}}{\partial \vec{y}} \vec{g}_y) \tag{6}$$

It can be easily verified by applying the above two mappings consecutively, 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. 14.

Listing 14: 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
```

4.2 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 embedding benchmark in Sec. A.2.

4.3 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. 5.1.

5 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 software like famous TensorFlow and PyTorch are not designed 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.

5.1 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 2 and Table 3. The Tapenade data is obtained by executing the docker file provided by the original benchmark, which provides a baseline for comparison.

NiLang's objective function is $\sim 2\times$ slower than normal code due to the uncomputing overhead. In this case, NiLang does not show advantage to Tapenade in obtaining gradients, the ratio between

# 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 2: 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.

computing the gradients and the objective function are close. This is because the bottleneck of this model is the matrix vector multiplication, traditional AD can already handle this function well. We emphasis the extra memory usage in NiLang is approximately 0.5% of the original program, this should be better than most existing AD frameworks.

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

NiLang performs the best on CPU, while having zero allocation during computing local Jacobians of size 15×2. We also compiled our adjoint program to GPU with no more than 10 lines of code with KernelAbstractions, which provides another $\sim 200\times$ speed up. Parallelizing the adjoint code requires the forward code not reading the same variable simultaneously in different threads, and this requirement is satisfied here.

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

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 and Xun Gao 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

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris

Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Jens Behrmann, David Duvenaud, and Jörn-Henrik Jacobsen. Invertible residual networks. *CoRR*, abs/1811.00995, 2018. URL http://arxiv.org/abs/1811.00995.

Charles H. Bennett. Logical reversibility of computation. 1973. URL https://ieeexplore.ieee.org/abstract/document/5391327.

Charles H. Bennett. Time/space trade-offs for reversible computation. *SIAM Journal on Computing*, 18(4):766–776, 1989. doi: 10.1137/0218053. URL https://doi.org/10.1137/0218053.

Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, M. Sohaib Alam, Shahnawaz Ahmed, Juan Miguel Arrazola, Carsten Blank, Alain Delgado, Soran Jahangiri, Keri McKiernan, Johannes Jakob Meyer, Zeyue Niu, Antal Száva, and Nathan Killoran. Pennylane: Automatic differentiation of hybrid quantum-classical computations, 2018. URL https://arxiv.org/abs/1811.04968.

Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: Unleashing julia on gpus. *CoRR*, abs/1712.03112, 2017. URL http://arxiv.org/abs/1712.03112.

Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing, 2012. URL https://arxiv.org/abs/1209.5145.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, Jan 2017. ISSN 1095-7200. doi: 10.1137/141000671. URL http://dx.doi.org/10.1137/141000671.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016. URL http://arxiv.org/abs/1604.06174.

Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation, 2014. URL https://arxiv.org/abs/1410.8516.

GAEL Forget, J-M Campin, Patrick Heimbach, Christopher N Hill, Rui M Ponte, and Carl Wunsch. Ecco version 4: An integrated framework for non-linear inverse modeling and global ocean state estimation. 2015. URL https://dspace.mit.edu/handle/1721.1/99660.

Michael P Frank. The r programming language and compiler. Technical report, MIT Reversible Computing Project Memo, 1997.

Michael P. Frank. Throwing computing into reverse. *IEEE Spectrum*, 54(9):32–37, Sep 2017. ISSN 0018-9235. doi: 10.1109/mspec.2017.8012237. URL http://dx.doi.org/10.1109/MSPEC.2017.8012237.

Michael Patrick Frank and Thomas F Knight Jr. *Reversibility for efficient computing*. PhD thesis, 1999. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.428.4962&rep=rep1&type=pdf.

Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU press, 2012.

Aidan N Gomez, Mengye Ren, Raquel Urtasun, and Roger B Grosse. The reversible residual network: Backpropagation without storing activations. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 30*, pp. 2214–2224. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/6816-the-reversible-residual-network-backpropagation-without-storing-activations.pdf.

Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and software*, 1(1):35–54, 1992. URL https://www.tandfonline.com/doi/abs/10.1080/10556789208805505.

Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

Jin-Guo Liu Hao Xie and Lei Wang. Automatic differentiation of dominant eigensolver and its applications in quantum physics. URL https://arxiv.org/abs/2001.04121.

Laurent Hascoet and Valérie Pascual. The tapenade automatic differentiation tool: Principles, model, and specification. *ACM Transactions on Mathematical Software (TOMS)*, 39(3):20, 2013. URL https://dl.acm.org/citation.cfm?id=2450158.

Tue Haulund. Design and implementation of a reversible object-oriented programming language, 2017. URL https://arxiv.org/abs/1707.07845.

Claudius Hubig. Use and implementation of autodifferentiation in tensor network methods with complex scalars, 2019. URL https://arxiv.org/abs/1907.13422.

Michael Innes. Don't unroll adjoint: Differentiating ssa-form programs, 2018. URL https://arxiv.org/abs/1810.07951.

Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. Fashionable modelling with flux, 2018. URL https://arxiv.org/abs/1811.01457.

Mike Innes, Alan Edelman, Keno Fischer, Christopher Rackauckas, Elliot Saba, Viral B. Shah, and Will Tebbutt. A differentiable programming system to bridge machine learning and scientific computing. *CoRR*, abs/1907.07587, 2019. URL http://arxiv.org/abs/1907.07587.

Li Jing, Yichen Shen, Tena Dubček, John Peurifoy, Scott Skirlo, Yann LeCun, Max Tegmark, and Marin Soljačić. Tunable efficient unitary neural networks (eunn) and their application to rnns, 2016. URL https://arxiv.org/abs/1612.05231.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. URL https://arxiv.org/abs/1412.6980.

J. G. Koller and W. C. Athas. Adiabatic switching, low energy computing, and the physics of storing and erasing information. In *Workshop on Physics and Computation*, pp. 267–270, Oct 1992. doi: 10.1109/PHYCMP.1992.615554. URL https://ieeexplore.ieee.org/document/615554.

Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM journal of research and development*, 5(3):183–191, 1961. URL https://ieeexplore.ieee.org/document/5392446.

Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. A theory of reversibility for erlang. *Journal of Logical and Algebraic Methods in Programming*, 100:71–97, Nov 2018. ISSN 2352-2208. doi: 10.1016/j.jlamp.2018.06.004. URL http://dx.doi.org/10.1016/j.jlamp.2018.06.004.

Robert Y Levine and Alan T Sherman. A note on bennett's time-space tradeoff for reversible computation. *SIAM Journal on Computing*, 19(4):673–677, 1990. URL https://epubs.siam.org/doi/abs/10.1137/0219046.

Hai-Jun Liao, Jin-Guo Liu, Lei Wang, and Tao Xiang. Differentiable programming tensor networks. *Physical Review X*, 9(3), Sep 2019. ISSN 2160-3308. doi: 10.1103/physrevx.9.031041. URL http://dx.doi.org/10.1103/PhysRevX.9.031041.

K. Likharev. Dynamics of some single flux quantum devices: I. parametric quantron. *IEEE Transactions on Magnetics*, 13(1):242–244, January 1977. ISSN 1941-0069. doi: 10.1109/TMAG.1977.1059351. URL https://ieeexplore.ieee.org/document/1059351.

Xiu-Zhe Luo, Jin-Guo Liu, Pan Zhang, and Lei Wang. Yao.jl: Extensible, efficient framework for quantum algorithm design, 2019. URL https://arxiv.org/abs/1912.10877.

Christopher Lutz. Janus: a time-reversible language. *Letter to R. Landauer.*, 1986.

Matthew MacKay, Paul Vicol, Jimmy Ba, and Roger B Grosse. Reversible recurrent neural networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 31*, pp. 9029–9040. Curran Associates, Inc., 2018. URL http://papers.nips.cc/paper/8117-reversible-recurrent-neural-networks.pdf.

Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In Francis Bach and David Blei (eds.), *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 2113–2122, Lille, France, 07–09 Jul 2015. PMLR. URL http://proceedings.mlr.press/v37/maclaurin15.html.

Ralph C. Merkle, Robert A. Freitas, Tad Hogg, Thomas E. Moore, Matthew S. Moses, and James Ryley. Mechanical computing systems using only links and rotary joints. *Journal of Mechanisms and Robotics*, 10(6), Sep 2018. ISSN 1942-4310. doi: 10.1115/1.4041209. URL http://dx.doi.org/10.1115/1.4041209.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017. URL https://openreview.net/forum?id=BJJsrmfCZ.

Kalyan S Perumalla. *Introduction to reversible computing*. Chapman and Hall/CRC, 2013.

Jarrett Revels, Miles Lubin, and Theodore Papamarkou. Forward-mode automatic differentiation in julia, 2016. URL https://arxiv.org/abs/1607.07892.

Chase Roberts, Ashley Milsted, Martin Ganahl, Adam Zalcman, Bruce Fontaine, Yijian Zou, Jack Hidary, Guifre Vidal, and Stefan Leichenauer. Tensornetwork: A library for physics and machine learning, 2019. URL https://arxiv.org/abs/1905.01330.

Matthias Seeger, Asmus Hetzel, Zhenwen Dai, Eric Meissner, and Neil D. Lawrence. Auto-differentiating linear algebra, 2017. URL https://arxiv.org/abs/1710.08717.

V. K. Semenov, G. V. Danilov, and D. V. Averin. Negative-inductance squid as the basic element of reversible josephson-junction circuits. *IEEE Transactions on Applied Superconductivity*, 13 (2):938–943, June 2003. ISSN 2378-7074. doi: 10.1109/TASC.2003.814155. URL https://ieeexplore.ieee.org/document/1211760.

Yan Shen and Yuxing Dai. Fast automatic differentiation for large scale bundle adjustment. *IEEE Access*, 6:11146–11153, 2018. URL https://ieeexplore.ieee.org/abstract/document/8307241/.

David Speyer and Bernd Sturmfels. Tropical mathematics. *Mathematics Magazine*, 82(3):163–173, 2009. URL https://www.tandfonline.com/doi/pdf/10.1080/0025570X.2009.11953615.

Filip Srajer, Zuzana Kukelova, and Andrew Fitzgibbon. A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning. *Optimization Methods and Software*, 33(4-6):889–906, 2018. URL https://openreview.net/forum?id=SMCWZLzTGDN.

Jun Takahashi and Anders W. Sandvik. Valence-bond solids, vestigial order, and emergent so(5) symmetry in a two-dimensional quantum magnet, 2020. URL https://arxiv.org/abs/2001.10045.

N Takeuchi, Y Yamanashi, and N Yoshikawa. Reversible logic gate using adiabatic superconducting devices. *Scientific reports*, 4:6354, 2014. URL https://www.nature.com/articles/srep06354.

Naoki Takeuchi, Yuki Yamanashi, and Nobuyuki Yoshikawa. Reversibility and energy dissipation in adiabatic superconductor logic. *Scientific reports*, 7(1):1–12, 2017. URL https://www.nature.com/articles/s41598-017-00089-9.

Clay S Turner. A fast binary logarithm algorithm [dsp tips & tricks]. *IEEE Signal Processing Magazine*, 27(5):124–140, 2010. URL http://www.claysturner.com/dsp/binarylogarithm.pdf.

Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. Openad/f: A modular open-source tool for automatic differentiation of fortran codes. *ACM Trans. Math. Softw.*, 34(4), July 2008. ISSN 0098-3500. doi: 10.1145/1377596.1377598. URL https://doi.org/10.1145/1377596.1377598.

Zhou-Quan Wan and Shi-Xin Zhang. Automatic differentiation for complex valued svd, 2019. URL https://arxiv.org/abs/1909.02659.

Weiqiang Zhu, Kailai Xu, Eric Darve, and Gregory C. Beroza. A general approach to seismic inversion with automatic differentiation, 2020. URL https://arxiv.org/abs/2003.06027.

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 4: Absolute runtimes in seconds for computing the objectives (O) and the backward pass (B) of sparse matrix operations. The matrix size is 1000×1000, and the element density is 0.05. The total time used in computing gradient can be estimated 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

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

$$\begin{aligned}\mathcal{L} = &\, \mathrm{Var}(\mathrm{dist}(L_1)) + \mathrm{Var}(\mathrm{dist}(L_2)) \\ &+ \exp(\mathrm{relu}(\overline{\mathrm{dist}(L_1)} - \overline{\mathrm{dist}(L_2)} + 0.1))) - 1\end{aligned} \tag{7}$$

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

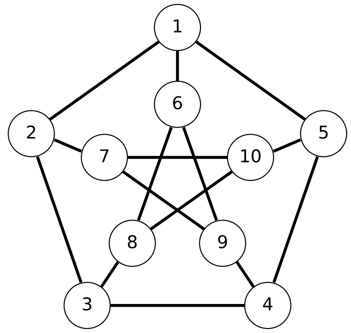

Figure 3: The Petersen graph has 10 vertices and 15 edges. We want to find a minimum embedding dimension for it.

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

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.

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 5: 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 5, 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 $k \geq 3$.

B CASES WHERE REVERSE COMPUTING SHOWS ADVANTAGE

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

Listing 15: 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 16: 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. 17

Listing 17: 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 18: 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
```