

First, thank all the referees for offering valuable suggestions to help to improve the writing of the paper. The referees generally agree that our paper is innovative in some aspects, but needs some improvement in the writing. We will keep improving our writing before the camera ready.

Some referees think our work lack the comparison to strong baselines like TensorFlow and PyTorch. This is not true, Tapenade is a very strong baseline in the field of **generic** AD. TensorFlow and PyTorch are **domain specific** AD software for traditional tensor-based machine learning. Some applications are not suited for tensors. e.g. People benchmarked different packages in the bundle adjustment application as shown in the figure. Tapenade is  $10^{3-5}$  times faster than PyTorch and TensorFlow. However Tapenade is commercial, close sourced and C/Fortran based. We are proud that NiLang is even better than Tapenade in this benchmark. We strongly recommend referees to read the ADBench paper [arXiv:1807.10129], you will find that we have chosen one of the worlds' best generic AD package as our baseline. We don't benchmark the popular Julia package Flux because we benchmarked its backend Zygote instead, where NiLang shows more than one order advantage in time.

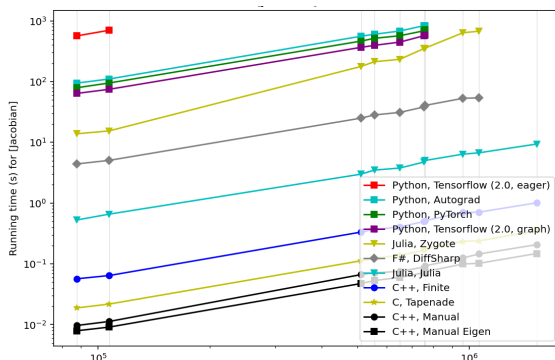


Figure 1: The bundle adjustment benchmark (conducted by the ADBench project of microsoft).

the Appendix. NiLang requires a user to preallocate a `branch_keeper` to store the decisions of branches to enforce reversibility inside the loop. However, if a user is writing it in free style, checkpointing has to insert stack operations inside the loop, which will slow down the program significantly. Not to say the optimal checkpointing itself is a well known hard topic. Allocating automatically for a user is even more dangerous in GPU programming. In NiLang, one can compile the bundle adjustment code to GPU to enjoy a 200x speed up with no more than 10 extra lines of code. It is much easier for users to completely avoid allocation inside a loop in NiLang comparing with checkpointing. Allocation can be the performance killer when one want to differentiate array mutation operations that many frameworks forbid such operations. NiLang is now getting more users, some use it to differentiate backtest policies and variational mean field computing. In both cases, NiLang help increase the performance by a factor of  $\sim 600$  comparing with Zygote, just because the array mutations is properly handled. As far as we know, computational graph is not convenient to represent inplace operations. Either the write-once `TensorArray` in TensorFlow or the mutable leaf tensors in PyTorch are not truly mutable. This explains one of the referees' concern that the computational graph can represent any program. Yes, but they are not convenient to describe array mutations.

Some referees want to know from which aspect NiLang is different from a traditional reversible programming language. For a long time, the reversible programming languages concern too much about theoretical elegance and ignored productivity. Researchers tried to add functional and object-oriented features to reversible languages. NiLang is special for that it supports many practical elements like arrays, complex numbers, fixed-point numbers, and logarithmic numbers and being an eDSL so that it can be used directly to accelerate the Zygote framework in Julia. And one of the referees is interested to know the limitations of NiLang. Reversible programming does not have any limitation in representing a computational process, because any program can be written in a reversible form. The most severe weakness of NiLang should be the floating-point arithmetic suffers from the rounding error. We can ease this problem by combining fixed-point numbers and logarithmic numbers, which will be explained in the next update.

2/4 referees think our result can not be reproduced. Considering both NiLang and the benchmarks are open source on GitHub, we don't think this comment is justified. We will address other comments like SVD is available in PyTorch and TensorFlow, comparing reversible programming and the reversible neural networks in the main text directly. Thanks for your valuable information.

At last, we encourage referees to view this project more from the "future" perspective. Nowadays, energy is becoming one of the most deadly bottlenecks for machine learning applications. From a physicist's perspective, we believe that reversible computing is the only correct approach to solve energy conundrums. Classical reversible computing has been silenced for  $\sim 15$  years, NiLang is trying to bridge the new trend machine learning and reversible computing. We will try our best to convey this point better in the updated version. As one of the referees said, it is a long overdue.