JaxDecompiler: Redefining Gradient-Informed Software Design

Pierrick Pochelu

University of Luxembourg, FSTM-DCS
pierrick.pochelu@uni.lu

Abstract. Among numerical libraries capable of computing gradient descent optimization, JAX stands out by offering more features, accelerated by an intermediate representation known as Jaxpr language. However, editing the Jaxpr code is not directly possible. This article introduces JaxDecompiler, a tool that transforms any JAX function into an editable Python code, especially useful for editing the JAX function generated by the gradient function. JaxDecompiler simplifies the processes of reverse engineering, understanding, customizing, and interoperability of software developed by JAX. We highlight its capabilities, emphasize its practical applications especially in deep learning and more generally gradient-informed software, and demonstrate that the decompiled code speed performance is similar to the original.

1 Introduction

Computational science and data science are witnessing a profound transformation, marked by the emergence of powerful new numerical frameworks. Recently, numerical frameworks such as Jax [2] [7], Tensorflow [1], PyTorch [4] and Sympy [13] have transformed the design of mathematical optimization by leveraging symbolic differentiation, eliminating the reliance on numerical approximations.

JAX distinguishes itself from PyTorch and TensorFlow by offering a broader range of functionalities. While PyTorch and TensorFlow primarily focus on deep neural network training, JAX not only integrates natively Autograd allowing which allows to address optimization problems with multi-order derivatives. The design of the function allows to use of the native Python language, including support for loops, indexing, and conditions. It provides an intuitive Numpy API [17] and includes distributed computing capabilities based on the MapReduce programming model [6].

The strength of JAX extends beyond its expressiveness; it also leverages a low-level Jaxpr code for faster execution. This efficiency can be further enhanced through Just-In-Time compilation using XLA [10]. However, it's crucial to acknowledge that the automatically generated Jaxpr code may not always be suitable for every scenario. This underscores the importance of a decompiler to translate the Jaxpr code into Python, facilitating modification before regenerating Jaxpr for final execution.

Many decompilers have been extensively proposed but they have mostly focused on languages like C, C++, and Java [8]. Those works show that the challenge of designing a decompiler is intrinsically linked to both the source and target language characteristics.

To bridge the gap between Jaxpr and Python and answer this technological gap, we introduce JaxDecompiler. It takes any JAX function as input and produces the equivalent Python code. The decompiler is required when the input function has been generated by a symbolic derivative in the Jaxpr language.

JaxDecompiler may serve a variety of purposes. Decompilers are generally used in applications such as malware detection [12], identifying duplicate code [14], and offering automatic code design recommendations [11]. JaxDecompilers is also useful for gradient code customization aiming to improve computing speed, and arithmetic stability, or export the gradient code for interoperability. PyTorch, Tensorflow, and JAX [1] allows already to customize gradient code by replacing the gradient code of a given function. JaxDecompiler enables another approach to do this, after the gradient code is computed with chain rule and exported as Python with the JaxDecompiler, the user may edit it and have full control of it.

Finally, JaxDecompiler provides the capability to export Python gradient code, facilitating interoperability with diverse software and platforms. In contrast, PyTorch and Tensorflow use neural network representation syntax for storing them such as ONNX [15] and TorchScript. This is constrained by the requirement for a dedicated neural network interpreter on the target platform and those representations are primarily tailored for neural networks. It's worth noting that, before the introduction of JaxDecompiler, JAX users typically converted their models into Tensorflow, and then from Tensorflow into ONNX for interoperability.

This paper is organized into four main sections. Section 2 provides practical examples of JaxDecompiler's usage. The inner working of the decompiler is presented in section 3. The speed performance of decompiled code is compared to the original JAX code on 3 applications in section 4. Finally, the conclusion in section 5 summarizes the significance and potential of JaxDecompiler and provides the GitHub link.

2 Step-by-step use cases

This section provides an example of the typical workflow usage of JaxDecompiler.

2.1 Step 1: JaxDecompiler input

Let's consider a typical JAX code below. In this example, we start with the function `jnp.log(1+jnp.exp(x))` and aim to obtain the derivative with respect to `x`

[1] URL: `https://jax.readthedocs.io/en/latest/notebooks/Custom_derivative_rules_for_Python_code.html`

encoded as float 32bits. The generated gradient function suffers from arithmetic instability for large `x` values greater than 87. JaxDecompiler addresses this issue by providing a decompiled version of the JAX function, allowing users to modify the code.

```
from jax import numpy as jnp
def f(x):
    return jnp.log(1+jnp.exp(x))

from jax import grad
gf=grad(f)
print(gf(100.)) # output: nan, expected: 1
```

The Jaxpr code of `gf` may be exported:

```
{ lambda ; a:f32[]. let
b:f32[] = exp a
c:f32[] = add 1.0 b
_:f32[] = log c
d:f32[] = div 1.0 c
e:f32[] = mul d b
in (e,) }
```

2.2 Step 2: Utilizing JaxDecompiler API

To address the limitations of the Jaxpr code and enable users to better understand, modify, and work with it, JaxDecompiler is introduced. The `Jaxpr2python` function is a key feature of JaxDecompiler, taking a JAX function as input and returning the decompiled function. The decompiled Python code is also provided as a string for exposing it to the user.

```
from JaxDecompiler import decompiler
gf2,py=decompiler.Jaxpr2python(gf,0.,is_python=True)
```

The main feature of `Jaxpr2python` function is to take a function as input (here `gf2`) and return the decompiled function (here `gf`) which behaves the same as the input one. The second argument (here `0.`) is a fake input used to specify the input type.

The `is_python` argument indicates that we return a second output, the decompiled function as a string (here `py`). Saving and importing `py` is identical to `gf`.

The decompiled Python code for the given example is as follows:

```
from jax.numpy import *
def gf2(a):
    b = exp(a)
    c = 1.0 + b
```

```
    _ = log(c)
    d = 1.0 / c
    e = d * b
    return e
```

It's important to emphasize that while the equivalence between Jaxpr and Python code may appear straightforward in this example, the JaxDecompiler may handle more intricate patterns, including conditional structures, distributed map operations, and loops. The GitHub repository link at the end provides the opportunity to explore and translate more complex Jaxpr code into comprehensive unit tests and applications, showcasing the decompiler's versatility and utility in handling a wide range of scenarios.

2.3 Step 3: Consuming JaxDecompiler output

The generated Python code is owned by the user, providing flexibility to use external code tools or manually edit the code. For example, the code can be manually improved for arithmetic stability with the if block statement to return an approximation:

```
from jax.numpy import *
def gf2(a):
    if a>87:
        return 1.
    b = exp(a)
    c = 1.0 + b
    _ = log(c)
    d = 1.0 / c
    e = d * b
    return e
```

This assembly-style language produced by decompilers is a well-known limitation for human maintenance of large software. Recent advancements in Large Language Models for processing decompiled codes [3] provide optimism for editing decompiler output into equivalent and more human-friendly code.

While the assembly-style Python produced by JaxDecompiler is a limitation when generating large software intended for human maintenance, it is advantageous for transpilation from Python to another language increasing interoperability. The transpiled code from Python into C is given below.

```
#include <cmath>
float gf2(float a) {
     if (a > 87) {
         return 1.0;
     }
     float b = exp(a);
     float c = 1.0 + b;
```

```
    float d = 1.0 / c;
    float e = d * b;
    return e;
}
```

3 JaxDecompiler design

This section presents the JaxDecompiler. First, an overview is given, and then the 3 main components are presented in more detail: Tokenizer, Line Translator, and Import Set.

3.1 Overview

The user gives the Jax function, and argument example (to automatically infer data types) and gets the decompiled Python code as output. The flow of data and processes for achieving this is depicted in Figure 1.

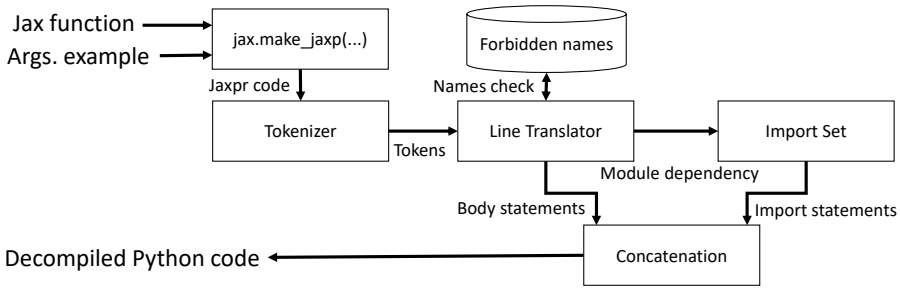

Fig. 1. Design of the JaxDecompiler. Edge represents data flow and the box the processing.

The first step is extracting the JaxPr code from the Jax function using `make_jaxpr` [2]. The Tokenizer splits Jaxpr lines into tokens, the Line Translator produces body statements, and the Import Set generates Python's import statements.

3.2 Tokenizer

Unlike compilers, decompilers take well-formatted code as input, making the input code trivial to analyze. The lexer step splits each Jaxpr statement into 4 sub-parts:

[2] `https://jax.readthedocs.io/en/latest/_autosummary/jax.make_jaxpr.html`

1. From 0 to n output variable name(s).
2. The operator.
3. The operator arguments (if any). It encompasses the arguments associated with the operator, if applicable.
4. From 0 to m input token(s) (variable names or literals).

Jaxpr is a strongly typed language but not the produced Python code. This is why the variable types are ignored at the tokenization step.

Some Jaxpr variable names may be forbidden by the Python language. A set of all forbidden variable names is stored (e.g. "if", "in", "is") to identify forbidden variable names and replace them with the uppercase version without the risk of colliding with the keywords.

3.3 Line Translator

JaxDecompiler's Line Translator contains a set of functions taking Jaxpr line represented as tokens and producing an equivalent Python string code. When a Jaxpr operator is missing, a clear Python Exception indicates the Jaxpr operator name to allow understanding and invite the community to implement it. Over 70 Jaxpr operators have been implemented, tested, and already addressing diversified applications.

We may enumerate some implemented Jaxpr operators in 3 categories:

- Element-wise: '+', '*', '-', '/', 'and', 'or', 'cos', 'sin', 'tan' ...
- Tensor manipulation: dot, transpose, convolution, sort ...
- High-order functions: condition, scan, parallel map, vectorized map ...

The Line Translator handles high-order instructions by applying recursive calls inside operator settings. In the Jaxpr language, high-order functions are represented using lambda expressions (unnamed function) stored in the operator arguments. To enhance code maintainability and reusability, a named function is produced and named with an incrementing index.

3.4 Import Set

Some Jaxpr functions can be translated into native Python language, while others require additional modules. Each time a line is encountered the Line Translator adds the import instruction's string into the import set. The Import Set keeps track of necessary import statements, ensuring efficiency and avoiding duplicate imports. We assume that the order of import has no importance.

Ultimately, after reading and translating all Jaxpr code, and the Import Set has tracked the necessary imports, the output of the Import Set and the Python code are concatenated. This constitutes the final Python code.

4 Performance of the decompiled code

The provided section gives an overview of the performance evaluation of code decompiled by JaxDecompiler. It aims to assess whether the decompiled Python code can maintain performance levels comparable to the original Jaxpr code. The performance is scrutinized across five distinct applications:

- Training with Multi-Layer Perceptrons (MLP): Involves computing the gradient of the neural network for backpropagation during training. Various settings are explored, including different numbers of data points, units per layer, and layers.
- Inference with MLP: Similar to the training scenario, performance is assessed while predicting MLPs under varying settings.
- Sorting using MapReduce: Utilizes multi-core CPU for sorting 32 million random numbers and returning the three smallest elements to the user. Specifically employs the multi-core "pmap."
- Molecular simulation: Involves simulating molecules represented as a 3D point cloud, utilizing gradient descent to update their positions at each time step to reach a stable equilibrium state.
- All reduce in multi-node multi-core settings: Computes an array of nine elements containing the average of process identifiers (MPI rank) based on the average allReduce collective communication operation in a distributed setting. Evaluated on the University of Luxembourg HPC [16] named Aion [3]. Notably, JAX users involved in data-parallel code use 'mpi4jax' [9], while JaxDecompiler translates this with 'mpi4py' [5].

For each application, the results are presented based on the average and standard deviation time (in seconds) across ten runs. The CPU used is an AMD EPYC with 128 cores (without hyper-threading) which is a common CPU in computing-intensive infrastructures. The performance is summarized in Table 1 for unjitted code and Table 2 after Just-In-Time (JIT) compilation.

The showcased applications underscore the versatility of decompiling various types of applications. In summary, the performance evaluation of the decompiled code demonstrates reasonable performance when compared to the original Jaxpr code. In the AllReduce scenario, the superior performance of the decompiled code is attributed to the direct nature of mpi4py in calling MPI (Message Passing Interface) primitives, as opposed to mpi4jax, which relies on mpi4py before reaching the MPI library. This additional layer of abstraction contributes to the observed performance differences. Additionally, the resilience in retaining the benefits of parallel and JIT instructions post-decompilation enhances the adaptability of the decompiled code for diverse performance-critical applications.

For the sake of transparency and reproducibility, we provide URLs at the document's end, offering access to JaxDecompiler's main code, the benchmarks used, and comprehensive tests. These resources serve as references for researchers and practitioners seeking to replicate and delve deeper into our study.

[3] `https://hpc-docs.uni.lu/systems/aion/`

Table 1. Performance comparison (seconds) of unjitted JAX function before and after decompilation

Application	Settings	JAX function	Decompiled
Training	1K points 1K units 2 layers	13.49 ± 0.5	$\mathbf{8.67 \pm 0.07}$
	16 points 1K units 128 layers	10.43 ± 0.15	$\mathbf{5.3 \pm 0.1}$
	16 points 8K units 2 layers	$\mathbf{8.85 \pm 0.06}$	11.52 ± 0.07
Inference	1K points 1K units 2 layers	$\mathbf{0.3685 \pm 0.001}$	0.4351 ± 0.0187
	16 points 1K units 128 layers	$\mathbf{0.4725 \pm 0.0008}$	0.635 ± 0.0405
	16 points 8K units, 2 layers	$\mathbf{0.4546 \pm 0.0002}$	0.4681 ± 0.0002
Sorting	1 core	$\mathbf{23.6494 \pm 0.2312}$	24.5791 ± 2.1247
	2 cores	11.778 ± 1.4836	$\mathbf{11.5528 \pm 1.3912}$
	16 cores	$\mathbf{1.0613 \pm 0.0381}$	1.0812 ± 0.0115
	128 cores	$\mathbf{0.2312 \pm 0.0246}$	0.2784 ± 0.0384
Physics	10,000 iter. 2 molecules	48.29 ± 0.06	$\mathbf{17.95 \pm 0.09}$
	1,000 iter. 20 molecules	92.04 ± 0.17	$\mathbf{33.99 \pm 0.04}$
AllReduce	1 node 128 MPI ranks	0.0439 ± 0.0025	$\mathbf{0.0021 \pm 0.0006}$
	4 nodes 512 MPI ranks	0.0749 ± 0.0099	$\mathbf{0.0199 \pm 0.0074}$
	16 nodes 2048 MPI ranks	0.0863 ± 0.0076	$\mathbf{0.0243 \pm 0.0084}$
	64 nodes 8192 MPI ranks	0.0998 ± 0.0108	$\mathbf{0.0285 \pm 0.0064}$

Table 2. Performance comparison (seconds) of JIT JAX function before and after decompilation

Application	Settings	JAX function	Decompiled
Training	1K points 1K units 2 layers	$\mathbf{6.3 \pm 0.35}$	6.48 ± 0.23
	16 points 1K units 128 layers	5.92 ± 0.05	$\mathbf{4.71 \pm 0.09}$
	16 points 8K units 2 layers	$\mathbf{11.06 \pm 0.04}$	12.38 ± 0.07
Inference	1K points 1K units 2 layers	$\mathbf{0.1378 \pm 0.0002}$	0.1622 ± 0.002
	16 points 1K units 128 layers	$\mathbf{0.4307 \pm 0.0023}$	0.5792 ± 0.0016
	16 points 8K units, 2 layers	$\mathbf{0.442 \pm 0.0001}$	0.6238 ± 0.0002
Sorting	1 core	24.1114 ± 1.1893	$\mathbf{23.7351 \pm 0.0911}$
	2 cores	$\mathbf{11.1945 \pm 1.1323}$	12.589 ± 4.1843
	16 cores	$\mathbf{1.5066 \pm 0.3904}$	1.5164 ± 0.3665
	128 cores	$\mathbf{1.6336 \pm 0.3475}$	1.706 ± 0.1575
Physics	10,000 iter. 2 molecules	$\mathbf{6.36 \pm 0.02}$	9.79 ± 0.03
	1,000 iter. 20 molecules	$\mathbf{11.9 \pm 0.02}$	18.63 ± 0.13
AllReduce	1 node 128 MPI ranks	$\mathbf{0.0006 \pm 0.0001}$	0.0021 ± 0.0001
	4 nodes 512 MPI ranks	0.0191 ± 0.0058	$\mathbf{0.0017 \pm 0.0001}$
	16 nodes 2048 MPI ranks	0.0247 ± 0.0049	$\mathbf{0.0177 \pm 0.0048}$
	64 nodes 8192 MPI ranks	0.0334 ± 0.0137	$\mathbf{0.0277 \pm 0.0043}$

5 Conclusion

In the ever-evolving landscape of numerical frameworks and gradient-informed software development, JAX has emerged as a versatile and performant framework. JaxDecompiler plays a pivotal role in reverse engineering machine learning functions generated by JAX, addressing a critical gap and empowering researchers to gain deeper insights into the inner workings of these functions. By offering a clearer and more accessible Python representation of Jaxpr code, JaxDecompiler facilitates debugging and analysis, crucial for identifying and addressing issues or unexpected behaviors. Furthermore, the software provides users with the capability to manually optimize the generated Python code, enhancing performance and arithmetic stability.

Notably, JaxDecompiler's performance aligns with that of code originally written, showcasing its effectiveness. While decompilers are inherently dependent on source and target language versions, JaxDecompiler stands as an open-source project, welcoming community contributions and remaining adaptable in the dynamic landscape of gradient-based software development and research.

Codes are available on GitHub: `https://github.com/PierrickPochelu/JaxDecompiler/`

Acknowledgment

The experiments presented in this paper were carried out using the HPC platform of the University of Luxembourg. Special thanks to Florian Felten for its review.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X.: Tensorflow: A system for large-scale machine learning. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. p. 265–283. OSDI'16, USENIX Association, USA (2016)
2. Bradbury, J., Frostig, R., Hawkins, P., Johnson, M.J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., Zhang, Q.: JAX: Autograd and XLA. Astrophysics Source Code Library, record ascl:2111.002 (Nov 2021)
3. Chen, Q., Lacomis, J., Schwartz, E.J., Goues, C.L., Neubig, G., Vasilescu, B.: Augmenting decompiler output with learned variable names and types. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 4327–4343. USENIX Association, Boston, MA (Aug 2022), `https://www.usenix.org/conference/usenixsecurity22/presentation/chen-qibin`
4. Collobert, R., Kavukcuoglu, K., Farabet, C.: Torch7: A matlab-like environment for machine learning. In: BigLearn, NIPS Workshop (2011)

5. Dalcin, L., Fang, Y.L.L.: mpi4py: Status update after 12 years of development. Computing in Science & Engineering **23**(4), 47–54 (2021). https://doi.org/10.1109/MCSE.2021.3083216
6. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (Jan 2008). https://doi.org/10.1145/1327452.1327492, https://doi.org/10.1145/1327452.1327492
7. Frostig, R., Johnson, M.J., Leary, C.: Compiling machine learning programs via high-level tracing. Systems for Machine Learning **4**(9) (2018)
8. Harrand, N., Soto-Valero, C., Monperrus, M., Baudry, B.: Java decompiler diversity and its application to meta-decompilation. Journal of Systems and Software **168**, 110645 (2020). https://doi.org/https://doi.org/10.1016/j.jss.2020.110645, https://www.sciencedirect.com/science/article/pii/S0164121220301151
9. Häfner, D., Vicentini, F.: mpi4jax: Zero-copy mpi communication of jax arrays. Journal of Open Source Software **6**(65), 3419 (2021). https://doi.org/10.21105/joss.03419, https://doi.org/10.21105/joss.03419
10. Leary, C., Wang, T.: Xla: Tensorflow, compiled! In: Tensorflow developer summit (2017)
11. Li, Z., Chen, T.H.P., Shang, W.: Where shall we log? studying and suggesting logging locations in code blocks. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. p. 361–372. ASE '20, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3324884.3416636, https://doi.org/10.1145/3324884.3416636
12. Mauthe, N., Kargén, U., Shahmehri, N.: A large-scale empirical study of android app decompilation. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 400–410 (2021). https://doi.org/10.1109/SANER50967.2021.00044
13. Meurer, A., Smith, C.P., Paprocki, M., Čertík, O., Kirpichev, S.B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J.K., Singh, S., et al.: Sympy: symbolic computing in python. PeerJ Computer Science **3**, e103 (2017)
14. Ragkhitwetsagul, C., Krinke, J., Clark, D.: Similarity of source code in the presence of pervasive modifications. In: 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 117–126 (2016). https://doi.org/10.1109/SCAM.2016.13
15. Sarah Bird, D.D.: ONNX. In: Workshop NIPS2017 (2017)
16. Varrette, S., Bouvry, P., Cartiaux, H., Georgatos, F.: Management of an academic hpc cluster: The ul experience. In: 2014 International Conference on High Performance Computing & Simulation (HPCS). pp. 959–967 (2014). https://doi.org/10.1109/HPCSim.2014.6903792
17. van der Walt, S., Colbert, S.C., Varoquaux, G.: The numpy array: A structure for efficient numerical computation. Computing in Science & Engineering **13**(2), 22–30 (2011). https://doi.org/10.1109/MCSE.2011.37