Compiling machine learning programs via high-level tracing

Roy Frostig*
Google Brain
frostig@google.com

Matthew James Johnson*
Google Brain
mattjj@google.com

Chris Leary
Google
leary@google.com

ABSTRACT

We describe JAX, a domain-specific tracing JIT compiler for generating high-performance accelerator code from pure Python and Numpy machine learning programs. JAX uses the XLA compiler infrastructure to generate optimized code for the program subroutines that are most favorable for acceleration, and these optimized subroutines can be called and orchestrated by arbitrary Python. Because the system is fully compatible with Autograd, it allows forward- and reverse-mode automatic differentiation of Python functions to arbitrary order. Because JAX supports structured control flow, it can generate code for sophisticated machine learning algorithms while maintaining high performance. We show that by combining JAX with Autograd and Numpy we get an easily programmable and highly performant ML system that targets CPUs, GPUs, and TPUs, capable of scaling to multi-core Cloud TPUs.

1 INTRODUCTION

Unlocking machine FLOPs has powered the explosion of progress in machine learning. Since the landmark work of AlexNet on dual-GPUs [5], the field has come a long way both in the number of FLOPs available to researchers and the ease with which these FLOPs can be harnessed. The JAX compiler aims to push further in this direction, enabling researchers to write Python programs—leaning on familiar and convenient libraries like Numpy [12] for numerics and Autograd [6] for automatic differentiation—that are automatically compiled and scaled to leverage accelerators and supercomputers.

The two goals of maximizing access to machine FLOPs and of facilitating research-friendly programmability are often in tension. Dynamic languages like Python offer convenient programming [1, 8] but are too unconstrained to enable optimized code generation. Meanwhile, effective hardware acceleration requires much more static information [13, 14]. Emerging supercomputer platforms like the NVIDIA DGX-1 and the Google Cloud TPU present new hardware capabilities that magnify the programmability challenge.

We can start to address this tension with an empirical observation: ML workloads often consist of large, accelerable, *pure-and-statically-composed* (PSC) subroutines orchestrated by dynamic logic. Roughly speaking, a function is pure when it does not have side effects. It is statically-composed, relative to a set of *primitive functions*, when it can be represented as a static data dependency graph on those primitives. Provided the primitive functions are themselves accelerable, e.g. when they comprise array-level numerical kernels and restricted control flow, PSC routines are prime candidates for acceleration: they delineate chunks of the original Python program from which all unused dynamism can be stripped out.

The JAX system is a just-in-time (JIT) compiler that generates code for PSC subroutines via *high-level tracing* together with the XLA compiler infrastructure. Tracing in JAX is high-level both in the sense (i) that it is implemented as user-level code within the source language, rather than as part of the source language's implementation, and (ii) that the trace primitives are not VM-level operations on basic data, but rather library-level numerical functions on array-level data, like matrix multiplies, convolutions, reductions along axes, elementwise operations, and multidimensional indexing and slicing [2, 9, 11].

```
import autograd.numpy as np
from autograd import grad
from jax import jit_ps

def predict(params, inputs):
  for W, b in params
    outputs = np.dot(inputs, W) + b
    inputs = np.tanh(outputs)
  return outputs

def loss(params, inputs, targets):
  preds = predict(params, inputs)
  return np.sum((preds - targets)**2)

grad_fun = jit_ps(grad(loss))  # Compiled gradient-of-loss function
```

Listing 1: A basic fully-connected neural network with JAX.

JAX is built atop the same tracing library used within Autograd, which, being designed for self-closure, recognizes its own operations as primitives. JAX also has Numpy's numerical functions among its primitives. As a result, it generates code for Python functions written in familiar Numpy and that involve arbitrary-order forward- and reverse-mode automatic differentiation. On the back end, JAX uses XLA for array-level program optimization and code generation. Whereas other systems focus on providing easy access to a fixed set of hand-written, target-specific numerical kernels, JAX provides a means of composition for all of XLA's supported target architectures: by trace-compiling PSC routines, JAX automatically stages out new kernels from existing ones.

The acronym JAX stands for "just after execution", since to compile a function we first monitor its execution once in Python.

2 SYSTEM DESIGN

The design of JAX is informed by the observation that ML workloads are typically dominated by PSC subroutines. In light of this observation, JAX *trace formation* simply requires users to annotate the PSC entry point; that is, a Python function for which the PSC assumption holds. This design exploits the property that these functions are often straightforward to identify in machine learning code, making them simple for the machine learning researcher to annotate with JAX's `jit_ps` decorator. While manual annotation presents a challenge for non-expert users and for "zero workload knowledge" optimization, it provides immediate benefits to experts and, as a systems research project, demonstrates the power of the PSC assumption. JAX *trace caching* creates a monomorphic signature for the parameters to the traced computation, such that

*Equal contributions

newly encountered array element types, array dimensions, or tuple members trigger a re-compilation.

On a trace cache miss, JAX executes the corresponding Python function and traces its execution into a graph of primitive functions with static data dependencies. Existing primitives comprise not only array-level numerical kernels, including Numpy functions and additional functions like convolutions and windowed reductions, but also restricted control flow functions, like a functional `while_loop` and cond (if-then-else). These control flow primitives are less familiar than syntactic Python control flow constructs, but they allow users to stage control flow into the compiled computation by preserving the PSC property. Finally, JAX includes some primitives for functional distributed programming, like `iterated_map_reduce`. The set of primitives is defined in Python and is extensible; new primitives must simply be annotated with a translation rule that builds corresponding XLA computations.

To generate code, JAX translates the trace into XLA HLO, an intermediate language that models highly accelerable array-level numerical programs. Broadly speaking, JAX can be seen as a system that lifts the XLA programming model into Python and enables its use for accelerable subroutines while still allowing dynamic orchestration. See Listing 2 for some example translation rules.

```
def xla_add(xla_builder, xla_args, np_x, np_y):
    return xla_builder.Add(xla_args[0], xla_args[1])

def xla_sinh(xla_builder, xla_args, np_x):
  b, xla_x = xla_builder, xla_args[0]
  return b.Div(b.Sub(b.Exp(xla_x), b.Exp(b.Neg(xla_x))), b.Const(2))

def xla_while(xla_builder, xla_args, cond_fun, body_fun, init_val):
  xla_cond = trace_computation(cond_fun, args=(init_val,))
  xla_body = trace_computation(body_fun, args=(init_val,))
  return xla_builder.While(xla_cond, xla_body, xla_args[-1])

jax.register_translation_rule(numpy.add,  xla_add)
jax.register_translation_rule(numpy.sinh, xla_sinh)
jax.register_translation_rule(while_loop, xla_while)
```

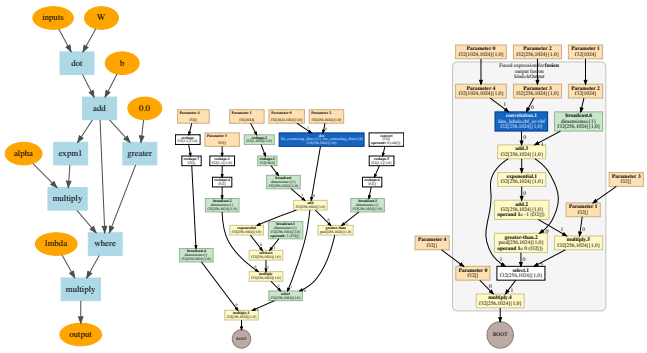
Listing 2: JAX translation rules from primitives to XLA HLO.

Finally, JAX is fully compatible with Autograd. See Listing 1, which compiles a gradient function for neural network training.

3 EXAMPLES AND EXPERIMENTS

Array-level fusion. To demonstrate the array-level code optimization and operation fusion that JAX and XLA provide, we compiled a single fully-connected neural network layer with SeLU nonlinearity [4] and show the JAX trace and XLA HLO graphs in Figure 1.

Truncated Newton-CG optimization on CPU. As a CPU benchmark, we implemented a truncated Newton-CG optimization algorithm, which performs approximate Newton-Raphson updates using a conjugate gradient (CG) algorithm in its inner loop. The inner CG algorithm is truncated when the residual norm is decreased below some threshold or a maximum number of inner iterations is reached. We compared the Python execution time with the JAX-compiled runtime on CPU using a single thread and several small example optimization problems, including a convex quadratic, a hidden Markov model (HMM) marginal likelihood, and a logistic regression. The XLA compile times were slow for some CPU examples but are likely to improve significantly in the future, and the speedups for the warmed-up code (Table 1) are substantial.



(a) JAX trace. (b) HLO before fusion. (c) HLO after fusion.

Figure 1: XLA HLO fusion for a layer with SeLU nonlinearity. The gray box indicates all ops are fused into the GEMM.

	Python	JAX	speedup
convex quadratic	4.12 sec	0.036 sec	114x
hidden Markov model fit	7.79 sec	0.057 sec	153x
logistic regression fit	3.62 sec	1.19 sec	3x

Table 1: Timing (sec) for Truncated Newton-CG on CPU.

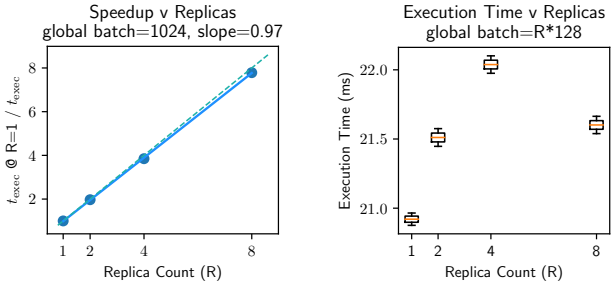


Figure 2: Scaling on a Cloud TPU for ConvNet training step.

Training a convolutional network on GPU. We implemented an all-conv CIFAR-10 network [10], involving only convolutions and ReLU activations. We JAX-compiled a single stochastic gradient descent (SGD) update step and called it from a pure Python loop, reporting the minimum average wall-clock step time across 100 trials in Table 2. As a reference, we implemented the same algorithm in TensorFlow and invoked it within a similar Python loop. The benchmarking environment was CUDA 8 driver 384.111 on an HP Z420 workstation.

	TF:GPU	JAX:GPU
t_{exec}	40.2 msec	41.8 msec
relative	1x	1.04x

Table 2: Timing (msec) for a JAX convnet step on GPU.

Cloud TPU scalability. JAX parallelization of global batch on Cloud TPU cores exhibits linear speedup (Figure 2, left). At fixed minibatch / replica, t_{exec} is minimally impacted by replica count (within 2ms, right). We used a Cloud TPU configuration with four chips and two cores per chip [3], so $R = 2$ is more efficient than $R = 8$, as on-chip communication is faster than between chips. The anomaly at $R = 4$ is due to an implementation detail of XLA's all-reduce.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, and others. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] Todd A Anderson, Hai Liu, Lindsey Kuper, Ehsan Totoni, Jan Vitek, and Tatiana Shpeisman. 2017. Parallelizing Julia with a Non-Invasive DSL. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 74. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[3] Jeff Dean. 2017. Machine Learning for Systems and Systems for Machine Learning. http://learningsys.org/nips17/assets/slides/dean-nips17.pdf. (2017).

[4] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. 2017. Self-Normalizing Neural Networks. *arXiv preprint arXiv:1706.02515* (2017).

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems (NIPS)*. 1097–1105.

[6] Dougal Maclaurin. 2016. *Modeling, Inference and Optimization with Composable Differentiable Procedures.* Ph.D. Dissertation. Harvard University.

[7] Dougal Maclaurin, David Duvenaud, Matthew Johnson, and Ryan P. Adams. 2015. Autograd: Reverse-mode differentiation of native Python. (2015). https://github.com/HIPS/autograd

[8] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. 2017. PyTorch. https://github.com/pytorch/pytorch. (2017).

[9] Jarrett Revels. 2017. Casette.jl. https://github.com/jrevels/Cassette.jl. (2017).

[10] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. 2014. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806* (2014).

[11] Ehsan Totoni, Todd A Anderson, and Tatiana Shpeisman. 2016. HPAT: high performance analytics with scripting ease-of-use. *arXiv preprint arXiv:1611.04934* (2016).

[12] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. 2011. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering* 13, 2 (2011), 22–30.

[13] Richard Wei, Lane Schwartz, and Vikram Adve. 2017. DLVM: A modern compiler framework for neural network DSLs. In *NIPS 2017 Autodiff Workshop*.

[14] XLA and TensorFlow teams. 2017. XLA — TensorFlow, compiled. https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html. (2017).