Compiling Machine Learning Programs via High-Level Tracing

JAX: A Domain-Specific Tracing JIT Compiler

Based on work by Roy Frostig, Matthew James Johnson, and Chris Leary
Group 8: Tongyuan Miao, Azfar Mohamed, Arnav Nikam, Tsubasa Okada, and Andrew Trybus

November 18, 2025

Motivations

Why Do We Need JAX?

- **The ML Performance Challenge**
  - Machine learning has exploded thanks to unlocking machine FLOPs
  - From AlexNet on dual-GPUs to modern supercomputers
  - Need to harness more computational power efficiently
- **The Programmability Tension**
  - Python: convenient, dynamic, research-friendly
  - Hardware acceleration: requires static information
  - Emerging platforms (NVIDIA DGX-1, Google Cloud TPU) magnify this challenge
- **The Solution**
  - JAX bridges this gap: write Python/Numpy code
  - Automatically compile and scale to accelerators
  - Maintain research-friendly programmability

Key Insight

Empirical Observation

ML workloads often consist of:

- **Large, accelerable subroutines** that are
- **Pure and statically-composed (PSC)**
- Orchestrated by **dynamic logic**

- These PSC routines are prime candidates for acceleration
- They allow us to strip out unused dynamism
- JAX targets these subroutines for optimization

Basic Terms & Concepts

Key Term XLA: Accelerated Linear Algebra

- **XLA (Accelerated Linear Algebra)**: Google's compiler infrastructure
  - Optimizes array-level numerical programs
  - Generates code for CPUs, GPUs, and TPUs
  - Performs operation fusion and other optimizations
- **JAX's Relationship to XLA**
  - JAX uses XLA as its backend compiler
  - Translates Python/Numpy code to XLA HLO (High-Level Operations)
  - XLA then optimizes and generates target-specific code

Key Concept: Pure and Statically Composed (PSC) Code

Pure Functions

Functions that have **no side effects**

- Same inputs always produce same outputs
- No global state modifications
- Predictable and testable

Statically Composed

Can be represented as a **static data dependency graph** on primitives

- Dependencies known at compile time
- No dynamic control flow (or restricted control flow)
- Graph structure doesn't change between runs

Example

Matrix multiplication, convolutions, elementwise operations, reductions

Key Term: Autograd and Automatic Differentiation

- **Autograd**: Automatic differentiation library for Python
  - Reverse-mode and forward-mode differentiation
  - Works with native Python and Numpy
  - Uses tracing to compute gradients
- **JAX's Compatibility**
  - Built on the same tracing library as Autograd
  - Fully compatible with automatic differentiation
  - Supports arbitrary-order forward- and reverse-mode AD
  - Can compile gradient functions!

Key Point

JAX = Numpy + Autograd + XLA compilation

Key Concept: High-Level Tracing

- **High-level** in two senses:
  1. Implemented as **user-level code** within Python
     - Not part of the language implementation
     - Can be extended and modified
  2. Trace primitives are **library-level operations**
     - Not VM-level operations on basic data
     - Array-level functions: matrix multiplies, convolutions, reductions
     - Multidimensional indexing and slicing
- **Just-In-Time (JIT) Compilation**
  - "JAX" = "Just After eXecution"
  - Monitor execution once in Python, then compile
  - Trace cache for monomorphic signatures

JAX System Design

High Level System Overview of JAX

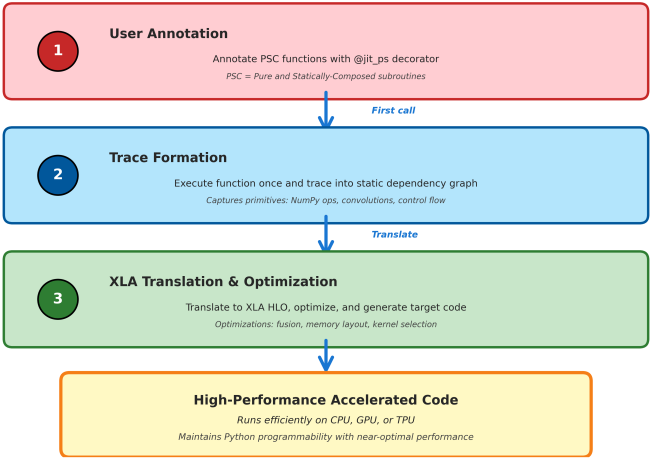
JAX: A Domain-Specific Tracing JIT Compiler

- Generates high-performance accelerator code
- From pure Python and Numpy ML programs
- Uses XLA compiler infrastructure
- Targets CPUs, GPUs, and TPUs

Design Philosophy

"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"

JAX System Workflow (Conceptual)



Primitives and Translation Rules

Primitive Functions

- Array-level numerical kernels (Numpy functions)
- Convolutions, windowed reductions
- Restricted control flow: `while_loop`, `cond` (if-then-else)
- Functional distributed programming: `iterated_map_reduce`

Extensibility

- Primitives defined in Python
- Extensible: new primitives need translation rules
- Translation rules map primitives to XLA HLO operations

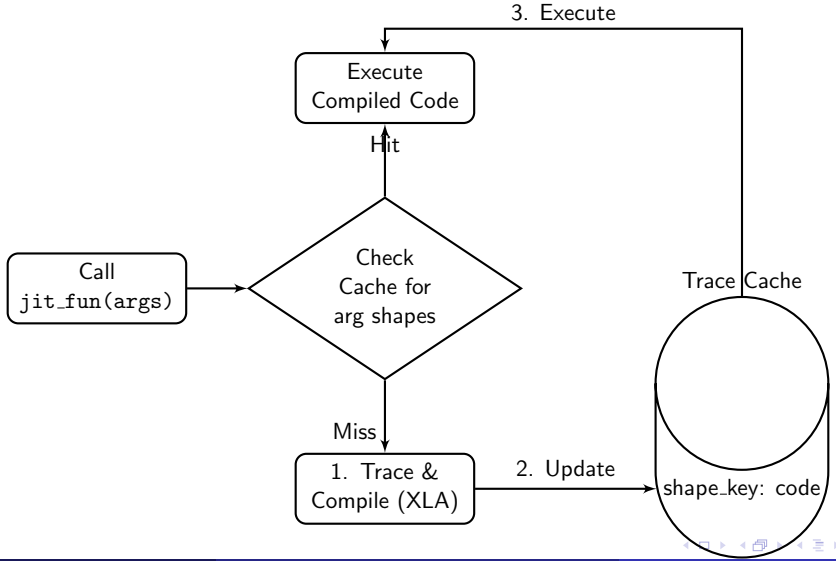
Example: Translation Rules

```
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))

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

Key Insight

Each primitive has a translation rule that builds corresponding XLA computations

Trace Caching



Autograd Integration

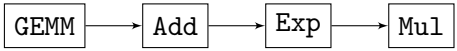
- **Full Compatibility**
  - Built on same tracing library as Autograd
  - Recognizes its own operations as primitives
  - Can compile gradient functions
- **Example Use Case**
  - Write neural network in Numpy
  - Use Autograd to get gradients
  - JAX compiles the gradient computation
  - Result: fast, compiled gradient computation

Examples & Results

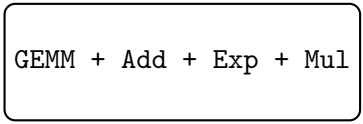
Array-Level Fusion

- **Example**: Fully-connected layer with SeLU nonlinearity
  - Multiple operations: dot, add, exp, greater, multiply, where
  - JAX traces these operations
  - XLA fuses them into optimized kernels

Before:



After:



Single Fused Kernel

Truncated Newton-CG on CPU

Problem	Python	JAX	Speedup
Convex quadratic	4.12 sec	0.036 sec	**114x**
HMM fit	7.79 sec	0.057 sec	**153x**
Logistic regression	3.62 sec	1.19 sec	**3x**

Key Observations

- Substantial speedups for warmed-up code
- XLA compile times can be slow (but improving)
- Best for compute-intensive operations

Convolutional Network on GPU

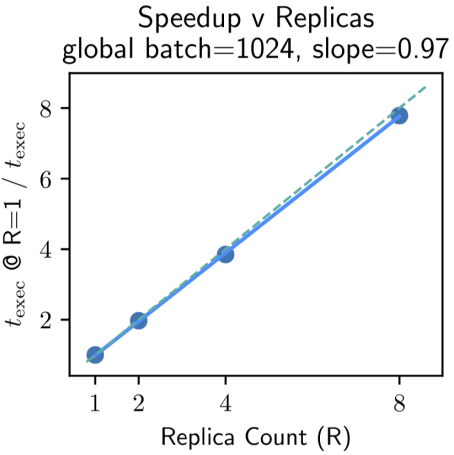
	TensorFlow	**JAX**
Execution time	40.2 msec	41.8 msec
Relative	1x	1.04x

Result

- Competitive performance with TensorFlow
- All-conv CIFAR-10 network
- Single SGD update step compiled

Cloud TPU Scalability

- **Linear Speedup**
  - Global batch parallelization across TPU cores
  - Speedup scales linearly with replica count
  - Slope = 0.97 (near-perfect scaling)
- **Execution Time Stability**
  - At fixed minibatch/replica, execution time minimally impacted
  - Within 2ms variation across replica counts
  - Efficient on-chip communication



Limitations

Limitations and Challenges

Manual Annotation Required

- Users must identify and annotate PSC entry points
- Challenge for non-expert users
- Requires understanding of what can be accelerated

Expert-Oriented System

- Most useful for ML researchers who understand the system
- "Zero workload knowledge" optimization not supported
- Niche application domain

Performance Limitations

- **Compile Time Overhead**
  - XLA compile times can be slow
  - Especially noticeable on CPU
  - Expected to improve in future
- **Cloud TPU Results**
  - Relatively little improvement on Cloud TPU
  - Some anomalies (e.g., R=4 due to XLA all-reduce implementation)
  - On-chip vs. between-chip communication matters
- **Workload-Specific**
  - Best for compute-intensive, PSC subroutines
  - Less benefit for dynamic, control-flow-heavy code

Trade-offs

Design Choices

- **Pros**: Immediate benefits for experts, demonstrates PSC power
- **Cons**: Not accessible to non-experts, requires manual work

Future Directions

- Automatic PSC detection
- Improved compile times
- Better TPU optimization
- More accessible interface

Conclusion

Summary

- **JAX** bridges Python programmability and hardware acceleration
- Uses **high-level tracing** to identify PSC subroutines
- Compiles to **XLA** for optimized code generation
- Fully compatible with **Autograd** for automatic differentiation
- Achieves **substantial speedups** (up to 153x) on appropriate workloads
- **Limitations**: Requires expert knowledge, manual annotation, niche application

Key Takeaways

The Big Idea

ML workloads have a structure we can exploit: large, pure, statically-composed subroutines that can be aggressively optimized

The System

JAX makes XLA's programming model accessible in Python while maintaining dynamic orchestration

The Result

Research-friendly Python code that scales to accelerators and supercomputers

Thank You!

Questions?