JAX
Compiling machine learning programs via high-level tracing

Differentiable Programming in ML and Scientific Computing

- Differentiable Programming
  - Auto gradient computation
- Application in Machine Learning
  - Gradients are crucial for model optimization
- Application in Scientific Computing
  - Gradients are crucial for optimization, estimation, etc

Motivation

- Limitations of Existing Frameworks
  - Pytorch, Tensorflow: no native support for various hardware (TPU)
  - Performance bottlenecks
- Challenges with Numerical Computing Libraries
  - Numpy: no native auto differentiation
  - Manual implementation of gradients
- Complexity of Hardware-Specific Optimization
  - Requires deep knowledge of hardware-specific optimizations

Overview

- Automatic Differentiation
- Functional Programming
  - Work with pure and statically-composed functions
- Interoperability with NumPy
- XLA Compilation
  - GPU
  - **TPU**
- 4 main transformations
  - grad(): automatically differentiate a function
  - vmap(): automatically vectorize operations
  - pmap(): parallel computation of SPMD programs
  - jit(): transform a function into a JIT-compiled version

Functional Programming

- Functional Programming
- Pure
  - No side effects
  - Referential transparency
- Statically-Composed
  - Static data dependency graph
  - A set of primitive functions

JAX Example

Computations are expressed as transformations on functions

```
1   import jax.numpy as jnp
2   from jax import grad
3   # Define a function
4   def square(x):
5       return x ** 2
6   # Compute the gradient of the function
7   grad_square = grad(square)
8   # Evaluate the gradient at x = 3
9   x = 3.0
10  gradient = grad_square(x)
11  print("Gradient at x =", x, ":", gradient)
```

Pytorch Comparison

Imperative programming, users define and execute computations dynamically

```python
import torch
# Define a tensor with requires_grad=True to track gradients
x = torch.tensor(3.0, requires_grad=True)
# Define the computation
square = x ** 2
# Compute the gradient
square.backward()
# Access the gradient
gradient = x.grad
print("Gradient at x =", x.item(), ":", gradient.item())
```

Tensorflow Comparison

Symbolic programming, users define computational graphs that represent mathematical operations

```
1  import tensorflow as tf
2  # Define a placeholder for the input
3  x = tf.placeholder(tf.float32)
4  # Define the computational graph
5  square = x ** 2
6  # Compute the gradient of the graph
7  grad_square = tf.gradients(square, x)
8  # Create a TensorFlow session and evaluate the gradient at x = 3
9  with tf.Session() as sess:
10     gradient = sess.run(grad_square, feed_dict={x: 3.0})
11     print("Gradient at x =", 3.0, ":", gradient[0])
```

Transformations: grad()

grad(): Automatically differentiate a function

```
import jax.numpy as jnp
from jax import grad
# Define a simple function
def f(x):
    return jnp.sin(x) + x**2
# Compute the gradient of the function using grad()
grad_f = grad(f)
# Evaluate the gradient at a specific point
x_value = 2.0
gradient_at_x = grad_f(x_value)
```

Transformations: vmap()

vmap(): Automatically vectorize operations

```python
import jax.numpy as jnp
from jax import vmap
# Define a function that squares each element in an array
def square(x):
    return x**2
# Create a batch of input arrays
batch = jnp.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Apply the square function to each array in the batch using vmap()
squared_batch = vmap(square)(batch)
```

Transformations: pmap()

pmap(): Enable parallel computation of Single Program, Multiple Data (SPMD) programs

```python
import jax.numpy as jnp
from jax import pmap
# Define a function that performs a matrix multiplication
def matmul(a, b):
    return jnp.dot(a, b)
# Parallelize the matrix multiplication across multiple devices
parallel_matmul = pmap(matmul)
# Create input matrices
A = jnp.ones((2, 2))
B = jnp.ones((2, 2))
# Perform parallel matrix multiplication
result = parallel_matmul(A, B)
```

Transformations: jit()

jit(): Transform a function into a Just-In-Time (JIT)-compiled version

```
1  import jax.numpy as jnp
2  from jax import jit
3  # Define a function that computes the sum of squares
4  def sum_of_squares(x):
5      return jnp.sum(x**2)
6  # Compile the function using jit()
7  compiled_fn = jit(sum_of_squares)
8  # Evaluate the compiled function
9  input_array = jnp.arange(10)
10 result = compiled_fn(input_array)
```

JAX tracing with Jaxprs: concepts

- Jaxpr
  - Intermediate representation
  - Use python interpreter to get statically-typed expressions
- Structure

```
jaxpr ::= { lambda Var* ; Var+.
            let Eqn*
            in  [Expr+] }
```

  - Parameters: constvars ; invars
  - Equations
  - Output

JAX tracing with Jaxprs: example

```
>>> from jax import make_jaxpr
>>> import jax.numpy as jnp
>>> def func1(first, second):
...    temp = first + jnp.sin(second) * 3.
...    return jnp.sum(temp)
...
>>> print(make_jaxpr(func1)(jnp.zeros(8), jnp.ones(8)))
{ lambda ; a:f32[8] b:f32[8]. let
    c:f32[8] = sin b
    d:f32[8] = mul c 3.0
    e:f32[8] = add a d
    f:f32[] = reduce_sum[axes=(0,)] e
  in (f,) }
```

JAX tracing with Jaxprs: handling control flow and function

```
>>> def func2(inner, first, second):
...   temp = first + inner(second) * 3.
...   return jnp.sum(temp)
...
>>> def inner(second):
...   if second.shape[0] > 4:
...     return jnp.sin(second)
...   else:
...     assert False
...
>>> def func3(first, second):
...   return func2(inner, first, second)
...
>>> print(make_jaxpr(func3)(jnp.zeros(8), jnp.ones(8)))
{ lambda ; a:f32[8] b:f32[8]. let
    c:f32[8] = sin b
    d:f32[8] = mul c 3.0
    e:f32[8] = add a d
    f:f32[] = reduce_sum[axes=(0,)] e
  in (f,) }
```

JAX tracing with Jaxprs: conditionals

```
>>> from jax import lax
>>>
>>> def func7(arg):
...   return lax.cond(arg >= 0.,
...                   lambda xtrue: xtrue + 3.,
...                   lambda xfalse: xfalse - 3.,
...                   arg)
...
>>> print(make_jaxpr(func7)(5.))
{ lambda ; a:f32[]. let
    b:bool[] = ge a 0.0
    c:i32[] = convert_element_type[new_dtype=int32 weak_type=False] b
    d:f32[] = cond[
      branches=(
        { lambda ; e:f32[]. let f:f32[] = sub e 3.0 in (f,) }
        { lambda ; g:f32[]. let h:f32[] = add g 3.0 in (h,) }
      )
      linear=(False,)
    ] c a
  in (d,) }
```

JAX tracing with Jaxprs: XLA_call

```
>>> from jax import jit
>>>
>>> def func12(arg):
...   @jit
...   def inner(x):
...     return x + arg * jnp.ones(1)  # Include a constant in the inner function
...   return arg + inner(arg - 2.)
...
>>> print(make_jaxpr(func12)(1.))
{ lambda ; a:f32[]. let
    b:f32[] = sub a 2.0
    c:f32[1] = pjit[
      name=inner
      jaxpr={ lambda ; d:f32[] e:f32[]. let
          f:f32[1] = broadcast_in_dim[broadcast_dimensions=() shape=(1,)] 1.0
          g:f32[] = convert_element_type[new_dtype=float32 weak_type=False] d
          h:f32[1] = mul g f
          i:f32[] = convert_element_type[new_dtype=float32 weak_type=False] e
          j:f32[1] = add i h
        in (j,) }
    ] a b
    k:f32[] = convert_element_type[new_dtype=float32 weak_type=False] a
    l:f32[1] = add k c
  in (l,) }
```

JAX tracing with Jaxprs: other higher-order primitives

- While
- Scan (loop over fixed size array)
- XLA_pmap

JAX - Introduction

- Just-in-time (JIT) compiler
- Convert pure Python and Numpy into high-performance code
- Run efficiently on various accelerators (CPUs, GPUs, TPUs)
- Write easily with Python while achieving significant speedups

JAX - Method

```
import numpy as np


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

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

- Only uses CPU
- No autodiff
- Not JIT compilation

JAX - Method

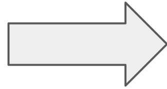
```
import numpy as np
import jax.numpy as np

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

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

- Could use GPU and TPU via XLA
- Autodiff
- JIT compilation
- Same API as Numpy

JAX - Method

Python function => JAX Intermediate Representation

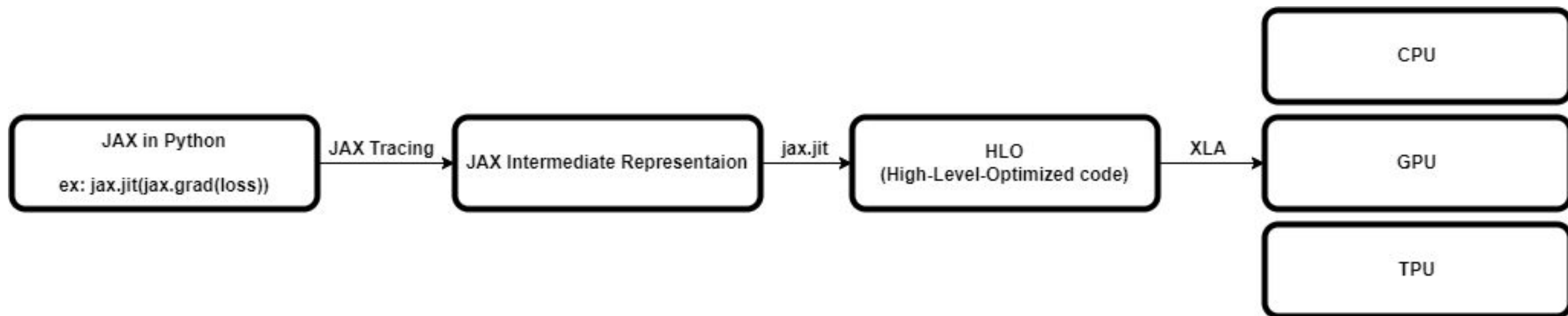
```python
from jax import lax

def log2(x):
  ln_x = lax.log(x)
  ln_2 = lax.log(2.0)
  return ln_x / ln_2
```



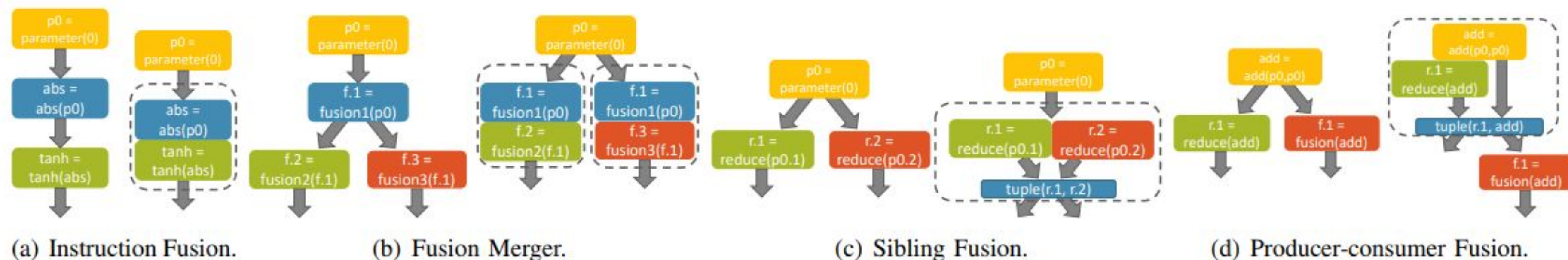
```
{ lambda  ; a.
  let b = log a
      c = log 2.0
      d = div b c
  in [d] }
```

- Autodiff
- JIT compilation
- Parallelization
- Batching

JAX - Design



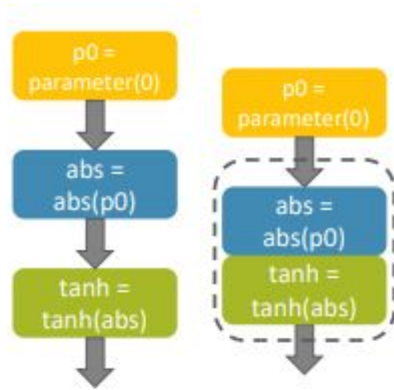
Operator Fusion Mechanisms in JAX (XLA)



(a) Instruction Fusion. (b) Fusion Merger. (c) Sibling Fusion. (d) Producer-consumer Fusion.

Fig. 1. Fusion strategies in XLA.

Instruction Fusion

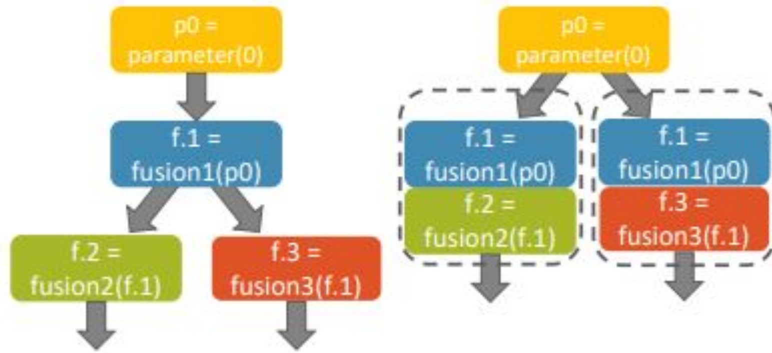


(a) Instruction Fusion.

Rules:

- Not expensive operations(e.g., convolution, sort, all reduce, etc.)
- Not too large for the GPU
- Not to exceed GPU hardware limits(e.g., threads per block, shared memory per block, etc.)

Fusion Merger



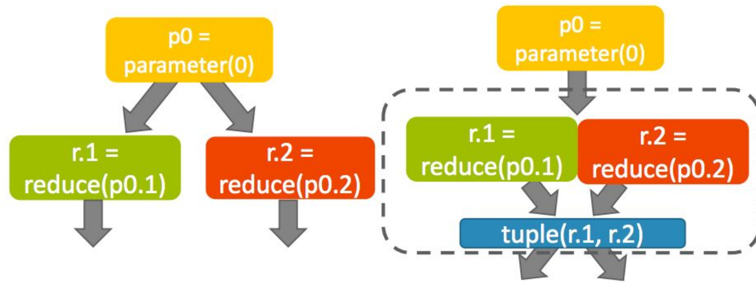
(b) Fusion Merger.

- Merge fusion instructions
- Reduce memory bandwidth requirements and kernel launch overhead

Rules:

- The fusion would not increase bytes transferred
- Producer operations are fusible with all consumers

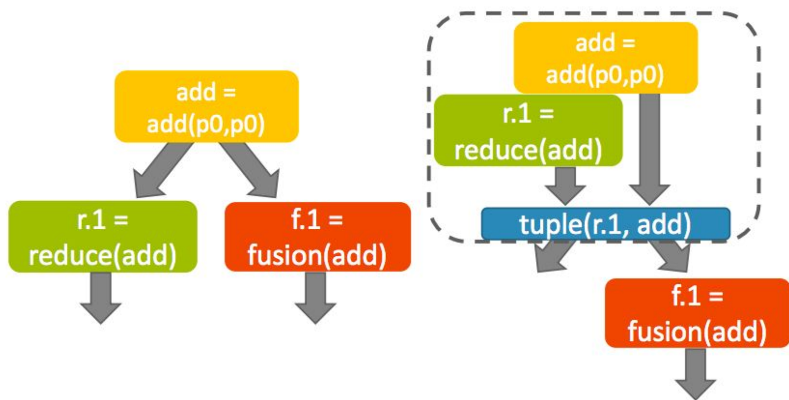
Sibling Fusion



(c) Sibling Fusion.

- Merge fusion instructions
- Reduce memory bandwidth requirements, because common input parameters have to be read only once

Producer-consumer Fusion



(d) Producer-consumer Fusion.

- Reduces memory bandwidth requirements by eliminating one read from memory

Rules:

- Sibling fusion and producer-consumer fusion can usually meet the fusion constraints at the same time. XLA will select the one that can give more fusion opportunities for later fusion optimizations

- Sibling has a higher priority over producer-consumer by default

Evaluation

Truncated Newton-CG optimization on CPU

- a CPU benchmark
- performs approximate Newton-Raphson updates using a conjugate gradient (CG) algorithm in its inner loop
- single thread
- on CPU

Evaluation

Speed up with example optimization problems

	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.

Evaluation

Training a convolutional network on GPU

- an all-conv CIFAR-10 network
- only convolutions and ReLU activations
- JAX-compiled a single stochastic gradient descent (SGD) update step
- Invoked from python code
- compared with TensorFlow
- CUDA 8 driver 384.111 on an HP Z420 workstation

Evaluation

Training a convolutional network on GPU

	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.

Evaluation

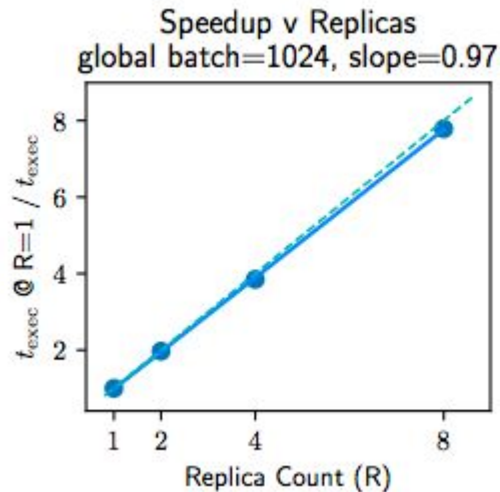
Cloud TPU scalability

- a Cloud TPU configuration with four chips and two cores per chip
- JAX parallelization of global batch on Cloud TPU cores exhibits linear speedup

Evaluation

Cloud TPU scalability

- JAX parallelization of global batch on Cloud TPU cores exhibits linear speedup



Evaluation

Cloud TPU scalability

- on-chip communication is faster than between chips