JAX Machine Learning Framework Presentation

Executive Summary

This presentation by Siwat (Customer Solutions Consultant at Google Thailand) provides a comprehensive introduction to JAX, Google's high-performance Python library for machine learning. The talk covers JAX's core concepts, performance advantages, and practical applications in modern AI development.

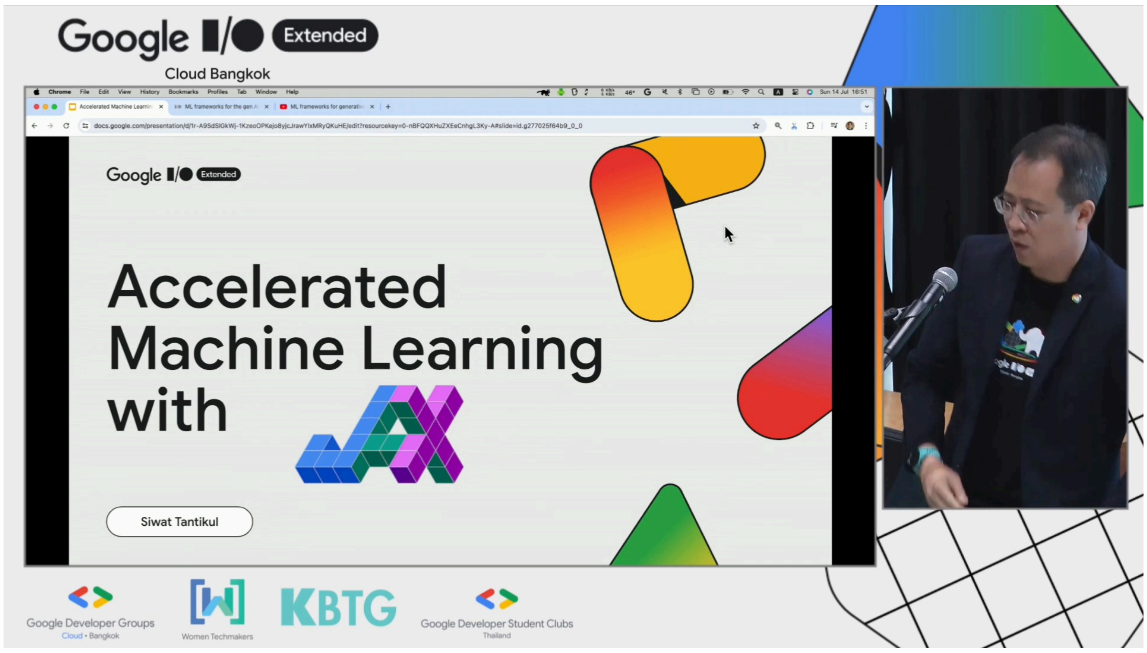
Speaker Highlights

Siwat - Google Customer Solutions Consultant

- Extensive experience with JAX implementation at Google
- Covers JAX usage from AlphaFold to Gemini
- Provides practical insights and real-world performance comparisons

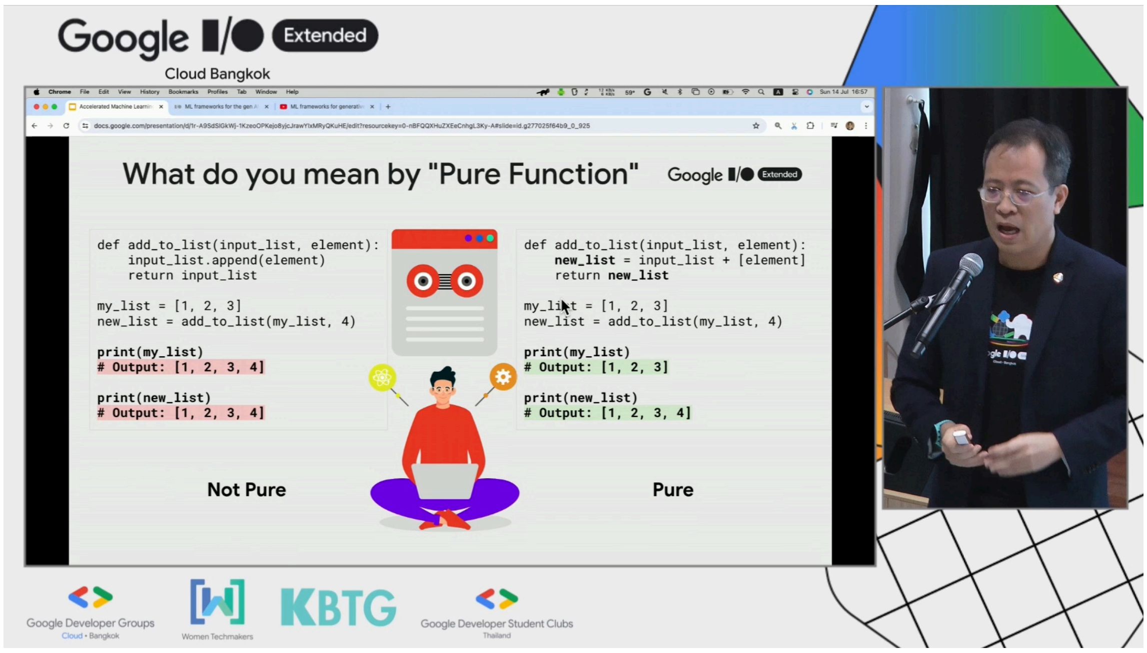
Timeline Summary

Introduction & Context (0-300s)



The presentation begins with an introduction to JAX and its significance in modern AI development. JAX has been used extensively at Google, from AlphaFold protein prediction to current Gemini models. The speaker emphasizes the growing computational requirements - from 10^15 petaflops for Transformer models in 2017 to over 20,000 billion petaflops for models like GPT-4.

Pure Functions & Functional Programming (375-465s)



What do you mean by "Pure Function"

```
def add_to_list(input_list, element):
    input_list.append(element)
    return input_list

my_list = [1, 2, 3]
new_list = add_to_list(my_list, 4)

print(my_list)
# Output: [1, 2, 3, 4]

print(new_list)
# Output: [1, 2, 3, 4]
```

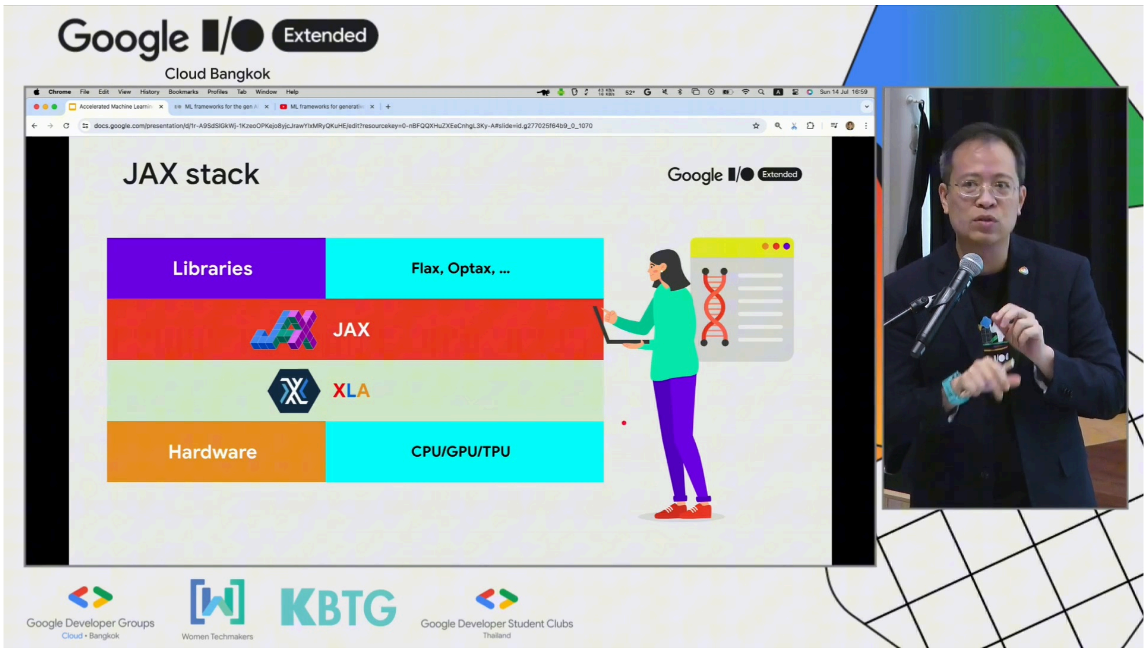
Not Pure

```
def add_to_list(input_list, element):
    new_list = input_list + [element]
    return new_list

my_list = [1, 2, 3]
new_list = add_to_list(my_list, 4)

print(my_list)
# Output: [1, 2, 3]

print(new_list)
# Output: [1, 2, 3, 4]
```

Pure

A critical concept in JAX is pure functional programming. Unlike traditional Python functions that may mutate input data, JAX requires immutable inputs. The speaker demonstrates how pure functions create copies rather than modifying original data, enabling better caching and optimization.

JAX Stack Architecture (462-646s)



JAX stack

Libraries	Flax, Optax, ...
JAX	
XLA	
Hardware	CPU/GPU/TPU

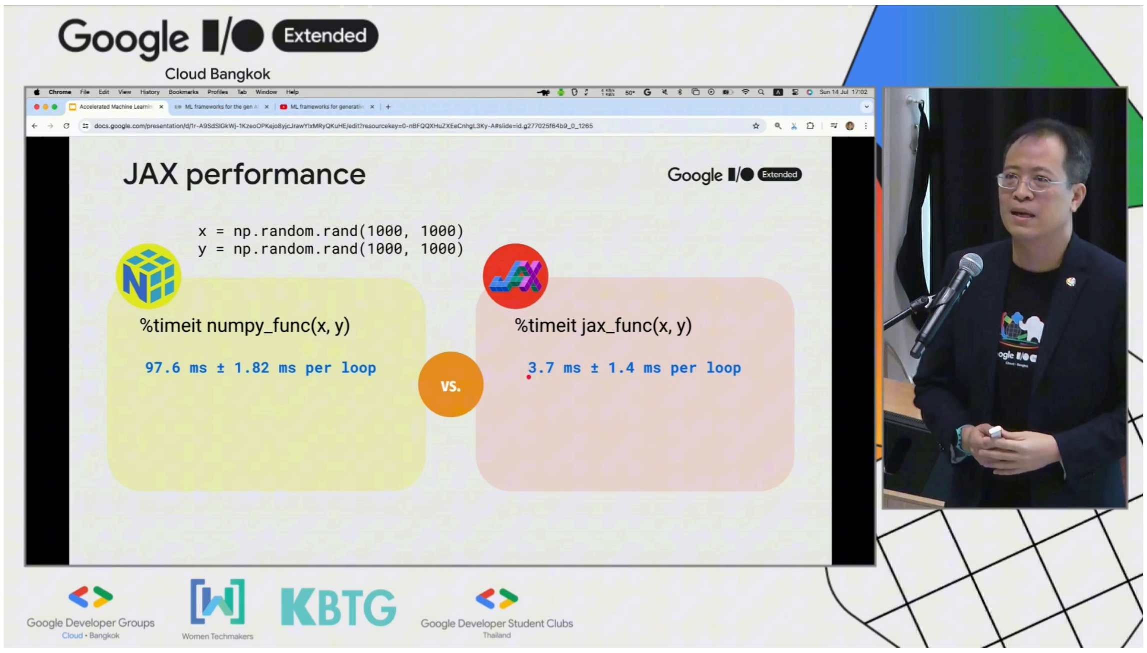
The JAX stack consists of:

- Hardware layer (CPU/GPU/TPU)
- XLA (Accelerated Linear Algebra) compiler
- JAX core library

- Ecosystem libraries (Flax, Optax, etc.)

XLA is the heart of JAX, converting Python code into optimized machine code for hardware acceleration.

Performance Comparison (649-780s)



JAX performance

```
x = np.random.rand(1000, 1000)
y = np.random.rand(1000, 1000)
```

%timeit numpy_func(x, y)

```
97.6 ms ± 1.82 ms per loop
```

vs.

%timeit jax_func(x, y)

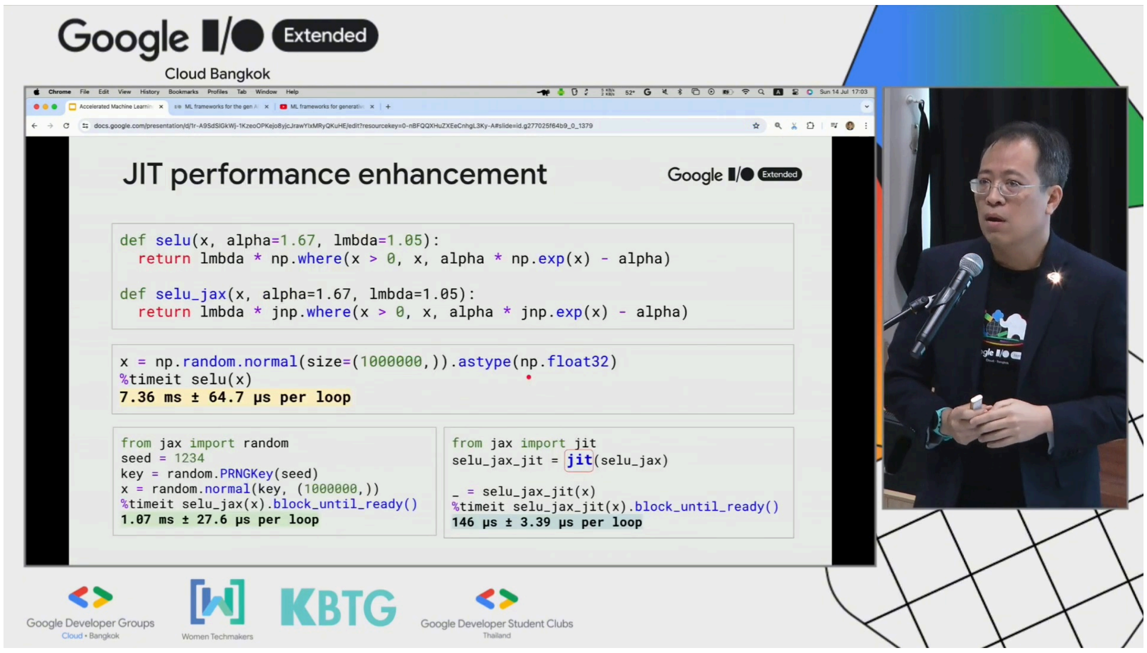
```
3.7 ms ± 1.4 ms per loop
```

Dramatic performance improvements demonstrated:

- NumPy: 97 milliseconds
- JAX: 3 milliseconds
- JAX with JIT: 146 microseconds

The speaker emphasizes that JAX excels with large-scale vectorized operations.

JIT Compilation (695-860s)



JIT performance enhancement

```
def selu(x, alpha=1.67, lmbda=1.05):
  return lmbda * np.where(x > 0, x, alpha * np.exp(x) - alpha)

def selu_jax(x, alpha=1.67, lmbda=1.05):
  return lmbda * jnp.where(x > 0, x, alpha * jnp.exp(x) - alpha)
```

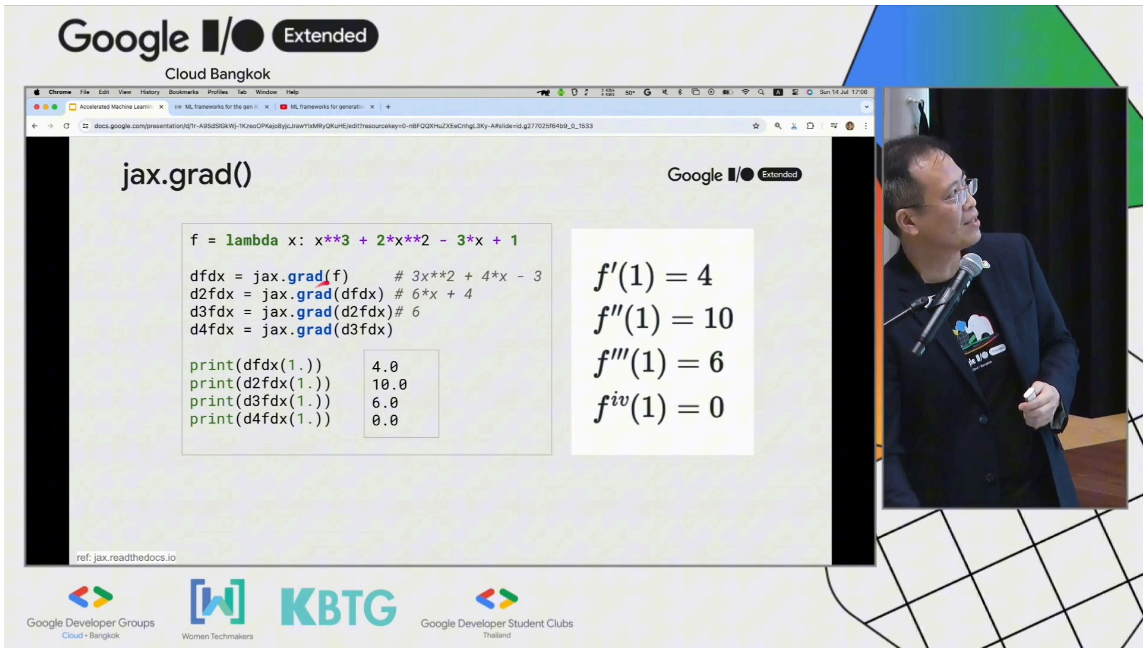
```
x = np.random.normal(size=(1000000,)).astype(np.float32)
%timeit selu(x)
7.36 ms ± 64.7 µs per loop
```

```
from jax import random
seed = 1234
key = random.PRNGKey(seed)
x = random.normal(key, (1000000,))
%timeit selu_jax(x).block_until_ready()
1.07 ms ± 27.6 µs per loop
```

```
from jax import jit
selu_jax_jit = jit(selu_jax)

_ = selu_jax_jit(x)
%timeit selu_jax_jit(x).block_until_ready()
146 µs ± 3.39 µs per loop
```

Just-In-Time (JIT) compilation is explained as a key performance booster. The first execution compiles the function, subsequent calls use cached compiled code. Simple decorator syntax (`@jit`) makes it easy to apply.

Auto-Differentiation (895-1019s)



jax.grad()

```
f = lambda x: x**3 + 2*x**2 - 3*x + 1

dfdx = jax.grad(f)       # 3x**2 + 4*x - 3
d2fdx = jax.grad(dfdx)  # 6*x + 4
d3fdx = jax.grad(d2fdx)# 6
d4fdx = jax.grad(d3fdx)

print(dfdx(1.))
print(d2fdx(1.))
print(d3fdx(1.))
print(d4fdx(1.))
```

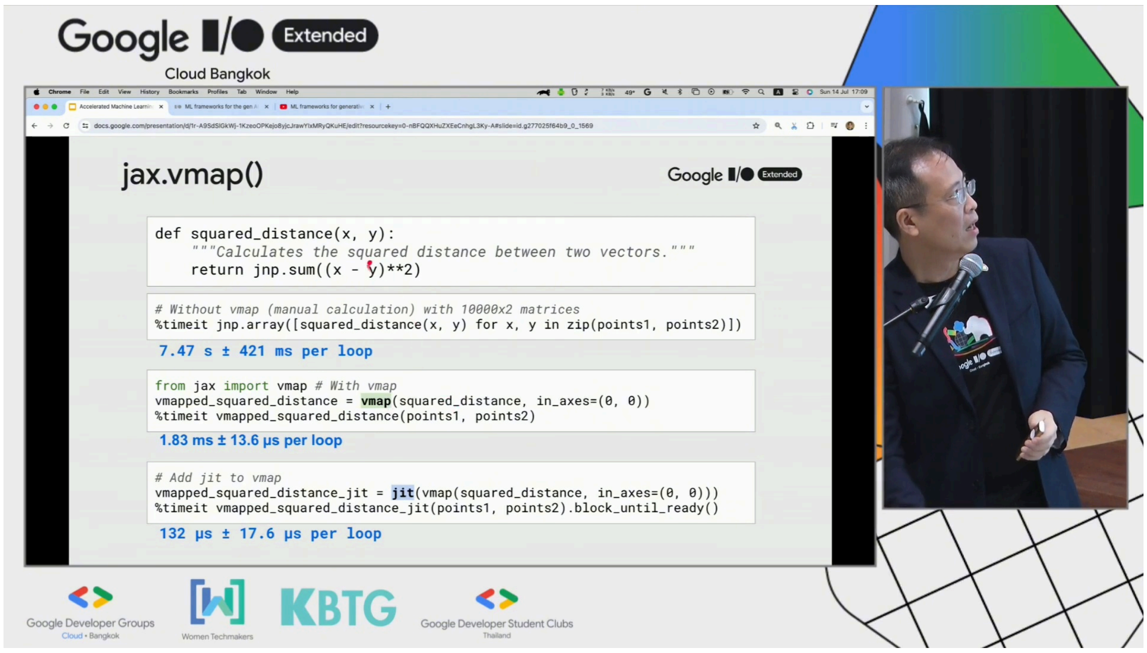
```
4.0
10.0
6.0
0.0
```

$$f'(1) = 4$$
$$f''(1) = 10$$
$$f'''(1) = 6$$
$$f^{iv}(1) = 0$$

ref: jax.readthedocs.io

JAX's automatic differentiation capabilities are showcased. The `jax.grad` function automatically computes derivatives regardless of equation complexity, supporting nested gradients of any degree. Practical example shown with logistic regression.

Vectorization with vmap (1019-1175s)



jax.vmap()

```
def squared_distance(x, y):
    """Calculates the squared distance between two vectors."""
    return jnp.sum((x - y)**2)
```

```
# Without vmap (manual calculation) with 10000x2 matrices
%timeit jnp.array([squared_distance(x, y) for x, y in zip(points1, points2)])
```

```
7.47 s ± 421 ms per loop
```

```
from jax import vmap # With vmap
vmapped_squared_distance = vmap(squared_distance, in_axes=(0, 0))
%timeit vmapped_squared_distance(points1, points2)
```

```
1.83 ms ± 13.6 µs per loop
```

```
# Add jit to vmap
vmapped_squared_distance_jit = jit(vmap(squared_distance, in_axes=(0, 0)))
%timeit vmapped_squared_distance_jit(points1, points2).block_until_ready()
```

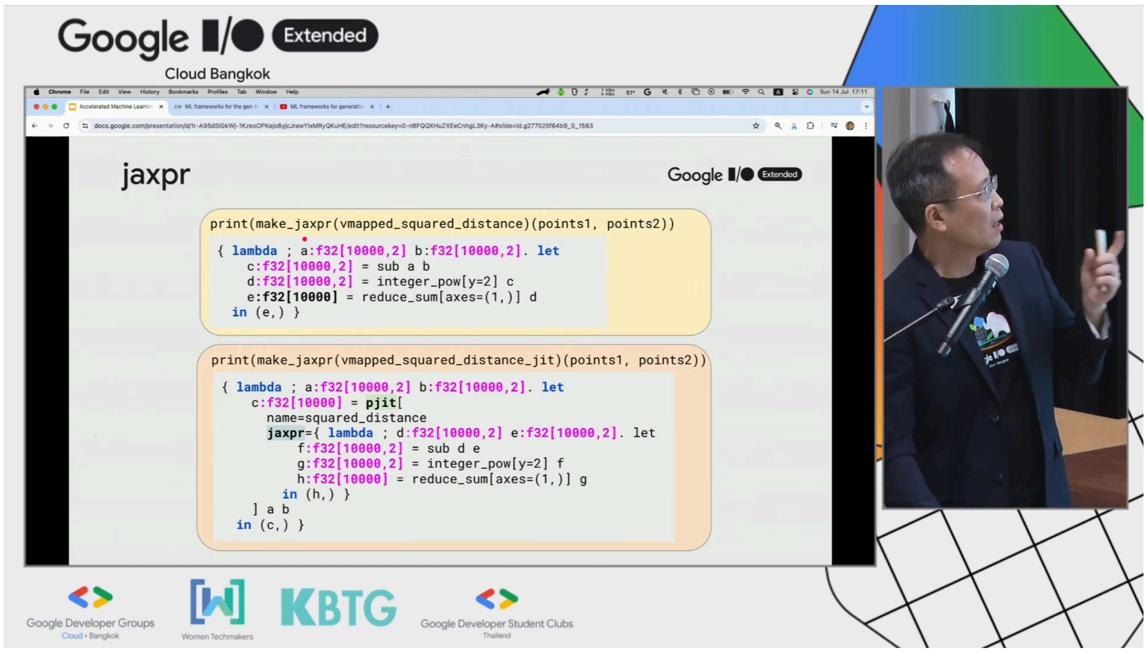
```
132 µs ± 17.6 µs per loop
```

Vector mapping (vmap) enables parallel operations across data batches:

- Traditional for-loop: 7 seconds
- With vmap: 1 second
- With vmap + JIT: 132 microseconds

This demonstrates massive speedups for batch processing operations.

XLA Internals (1180-1302s)



jaxpr

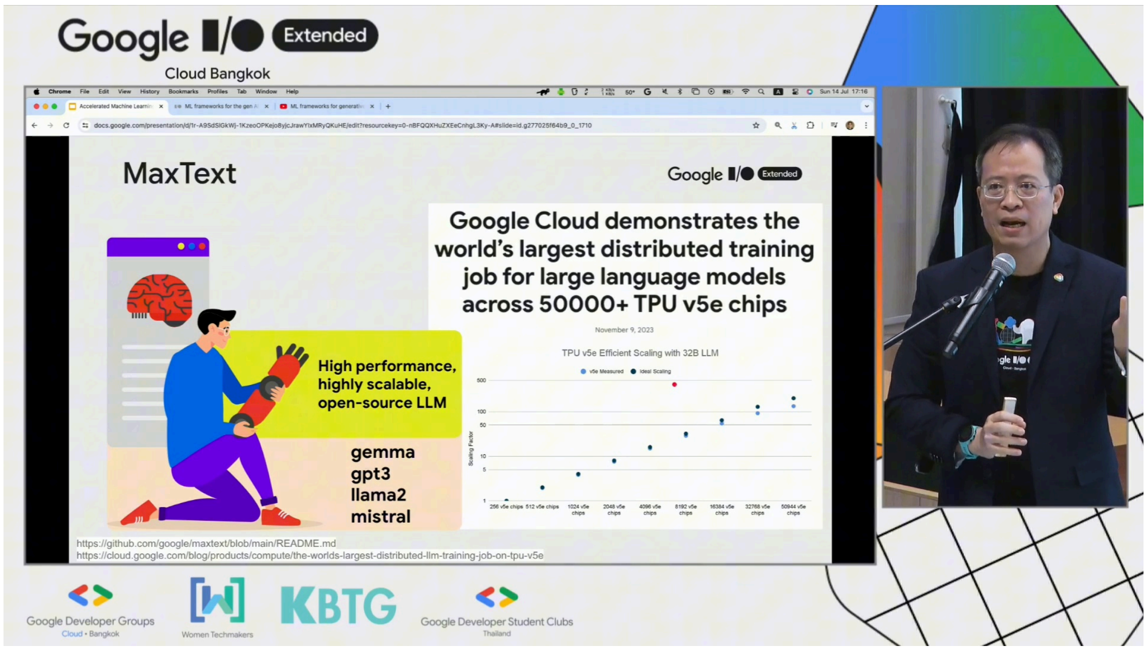
```
print(make_jaxpr(vmapped_squared_distance)(points1, points2))

{ lambda ; a:f32[10000,2] b:f32[10000,2]. let
    c:f32[10000,2] = sub a b
    d:f32[10000,2] = integer_pow[y=2] c
    e:f32[10000] = reduce_sum[axes=(1,)] d
  in (e,) }
```

```
print(make_jaxpr(vmapped_squared_distance_jit)(points1, points2))

{ lambda ; a:f32[10000,2] b:f32[10000,2]. let
    c:f32[10000] = pjit[
      name=squared_distance
      jaxpr={ lambda ; d:f32[10000,2] e:f32[10000,2]. let
          f:f32[10000,2] = sub d e
          g:f32[10000,2] = integer_pow[y=2] f
          h:f32[10000] = reduce_sum[axes=(1,)] g
        in (h,) }
    ] a b
  in (c,) }
```

Deep dive into how JAX converts Python code through multiple compilation stages:

1. JAX expressions (jaxpr)
2. StableHLO intermediate representation
3. XLA optimization
4. Machine code for hardware

Neural Networks & MaxText (1411-1615s)



MaxText

Google Cloud demonstrates the world's largest distributed training job for large language models across 50000+ TPU v5e chips

November 9, 2023

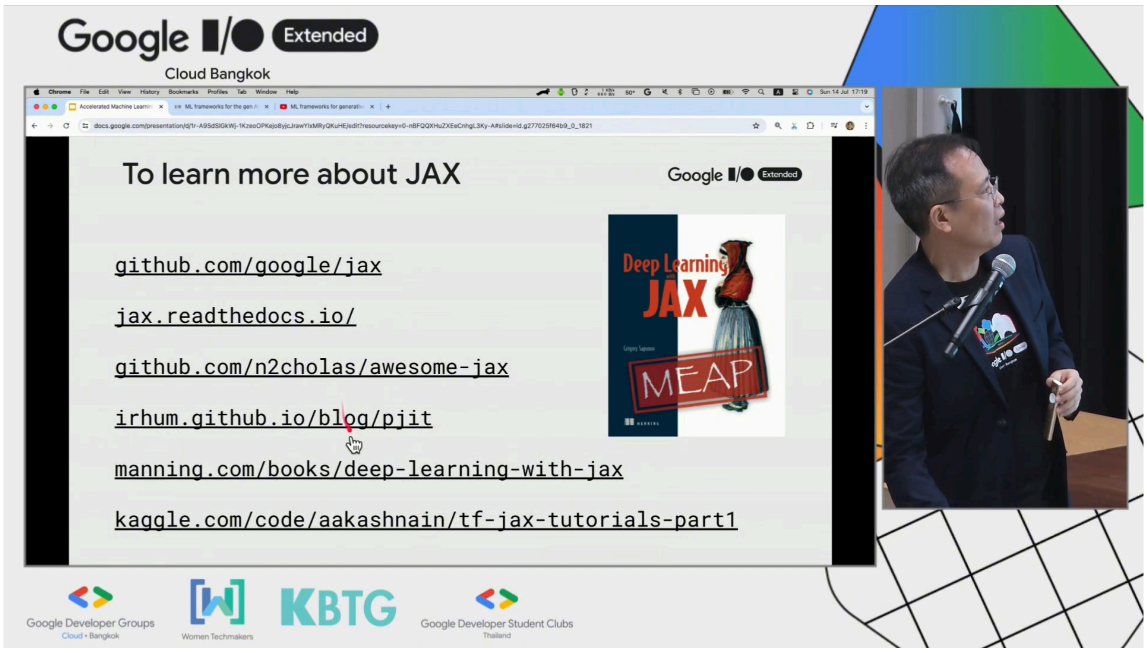
High performance, highly scalable, open-source LLM

gemma
gpt3
llama2
mistral

https://github.com/google/maxtext/blob/main/README.md
https://cloud.google.com/blog/products/compute/the-worlds-largest-distributed-llm-training-job-on-tpu-v5e

JAX ecosystem for neural networks includes Flax and Equinox libraries. MaxText is introduced as Google's open-source framework for training large language models, successfully used to train models with 50,000 TPUs achieving 55% model FLOP utilization.

Getting Started Resources (1640-1776s)


To learn more about JAX

github.com/google/jax

jax.readthedocs.io/

github.com/n2cholas/awesome-jax

irhum.github.io/blog/pjit

manning.com/books/deep-learning-with-jax

kaggle.com/code/aakashnain/tf-jax-tutorials-part1

Recommended learning resources:

- JAX tutorials on Kaggle (9 lessons)
- Awesome JAX repository
- JAX ReadTheDocs official documentation
- Google's JAX GitHub
- Book by Grigory Sapunov for deep understanding

Key Takeaways

- JAX provides NumPy-compatible syntax with dramatic performance improvements
- Core features: JIT compilation, auto-differentiation, vectorization, parallelization
- Used extensively at Google for production AI systems
- Open-source with strong ecosystem (Flax, Optax, MaxText)
- Ideal for high-performance machine learning and large-scale model training
- Functional programming paradigm enables better optimization and caching
- Performance gains of 100-1000x possible with proper usage of JIT and vmap