= JAX – Moderní nástroj pro diferenciální programování a výpočetní akceleraci =

TOC

== Co je JAX? == JAX je open-source knihovna od Google, která kombinuje NumPy-like API s automatickým diferenciálním výpočtem a kompilací na různé hardwarové akcelerátory (CPU, GPU, TPU). Díky funkci jit (just-in-time) a transformacím jako grad, vmap, pmap a jit umožňuje psát čistý, funkcionální kód a nechat JAX optimalizovat a distribuovat výpočty.

== Historie a vývoj == | ^ Rok ^ | ^ Událost ^ | | 2018 | První veřejná verze JAX (v0.1) – zaměřená na automatické diferencování pomocí Autograd. | | 2019 | Přidání XLA (Accelerated Linear Algebra) backendu, podpora GPU a TPU. | | 2020 | Vydání v0.2 – zavedení jit, vmap a pmap. | | 2021 | JAX 1.0 – stabilní API, rozšířená podpora pro float8, bfloat16, a mixed-precision. | | 2022 | Integrace s Flax, Haiku a Optax – ekosystém pro modelování, trénink a optimalizaci. | | 2023 | JAX 2.0 – podpora GPU-TensorCore a TPU-v5 instrukcí, rozšířené pjit a sharding. | | 2024 | Rozšířený JAX 3.0 preview – podpora distributed arrays (GDA) a sparse operací. |

== Základní koncepty == === 1. Funkcionální programování === JAX vyžaduje, aby funkce byly čisté (bez vedlejších efektů). To umožňuje:

```
  Transformace (grad, jit, vmap, pmap, pmap, pjit) – mohou být aplikovány na libovolnou funkci.
  Deterministické chování – důležité pro reprodukovatelnost experimentů.
```

2. Autodiff (automatické diferenciace)

```
  grad(f) – vrací funkci, která počítá gradient skalárního výstupu.
  value_and_grad(f) – vrací jak hodnotu, tak gradient.
  jacfwd, jacrev – Jacobian pomocí forward- nebo reverse-mode.
```

=== 3. Just-In-Time kompilace (JIT) === jit(f) kompiluje funkci pomocí XLA a uloží optimalizovaný kód pro daný hardware. Při opakovaném volání je výkon až 10-30× rychlejší než čistý NumPy.

=== 4. Vektorizace (vmap) === vmap(f) automaticky mapuje funkci přes batche, čímž eliminuje potřebu explicitních for-loopů a umožňuje SIMD-styl výpočty.

5. Paralelní a distribuovaná výpočty (pmap, pjit)

```
  pmap(f) – paralelní mapování přes více zařízení (GPU/TPU) na jednom hostu.
  pjit(f, in_sharding, out_sharding) – explicitní sharding pro distribuované výpočty napříč clusterem.
```

== Instalace ==

```
Instalace z PyPI (CPU-only)

pip install --upgrade "jax[cpu]"
Instalace s GPU (CUDA 12)
```

```
pip install --upgrade "jax[cuda12_cudnn89]" -f
https://storage.googleapis.com/jax-releases/jax_cuda_releases.html
Instalace s TPU (v Cloud Shell)

pip install --upgrade "jax[tpu]" -f https://storage.googleapis.com/jax-
releases/libtpu_releases.html
```

== Jednoduchý příklad – lineární regrese ==

```
 import jax.numpy as jnp from jax import grad, jit, random
Generování syntetických dat

key = random.PRNGKey(0) X = random.normal(key, (1000, 3)) true_w =
jnp.array([1.5, -2.0, 0.7]) y = X @ true_w + 0.1 * random.normal(key,
(1000,))
Definice ztrátové funkce

def loss(w, X, y): preds = X @ w return jnp.mean((preds - y) ** 2)
Gradient a JIT-ovaná verze

grad_loss = jit(grad(loss))
Optimalizace (SGD)

def sgd_step(w, lr, X, y): g = grad_loss(w, X, y) return w - lr * g
Trénink

w = jnp.zeros(3) lr = 0.01 for epoch in range(500): w = sgd_step(w, lr, X,
y)

print("Naučené váhy:", w)
```

Co se zde děje?

```
  Používáme jax.numpy (drop-in replacement za NumPy).
  jit kompiluje gradientní výpočet jednou a pak ho znovu používá.
  Všechny operace jsou vektorové a běží na GPU/TPU, pokud jsou k dispozici.
```

Pokročilé funkce

=== 1. Vmap – batched inference ===

```
 def model(params, x): w, b = params return jnp.dot(x, w) + b

batched_model = jax.vmap(model, in_axes=(None, 0))

params = (jnp.array([0.2, -0.5]), 0.1) xs = jnp.arange(10).reshape(5, 2) # 5
příkladů, 2 dimenze print(batched_model(params, xs))
```

=== 2. Pmap – data-parallel trénink na 8 TPU ===

```
 from jax import pmap

def step(state, batch): grads = grad(loss)(state['params'], batch['X'],
batch['y']) new_params = jax.tree_util.tree_map(lambda p, g: p - 0.001 * g,
state['params'], grads) return {'params': new_params}, None
Rozdělení dat na 8 zařízení

def shard(data): return data.reshape(8, -1, *data.shape[1:])

state = {'params': jnp.zeros((3,))} batch = {'X': shard(X), 'y': shard(y)}
state, _ = pmap(step, axis_name='devices')(state, batch)
```

=== 3. Pjit a sharding – distribuované matice ===

```
 from jax.experimental import pjit from jax.sharding import Mesh,
PartitionSpec as P

mesh = Mesh(jax.devices(), ('data',)) @pjit def matmul(A, B): return
jnp.dot(A, B)

A = jnp.arange(40964096).reshape(4096, 4096) B =
jnp.arange(40964096).reshape(4096, 4096)
Sharding specifikace

A_sharded = jax.device_put(A, jax.sharding.NamedSharding(mesh, P('data',
None))) B_sharded = jax.device_put(B, jax.sharding.NamedSharding(mesh,
P(None, 'data')))

C = matmul(A_sharded, B_sharded)
```

JAX a TPU – ideální dvojice

```
  XLA backend: JAX automaticky překládá operace do XLA, což je
optimalizovaný kompilátor používaný i v Google TPU.
  bfloat16 a float8: Podpora těchto formátů umožňuje vyšší throughput a
nižší spotřebu paměti na TPU.
  Sparsity: V JAX 3.0 je experimentální podpora pro sparse matice, což je
důležité pro velké jazykové modely (LLM).
```

Příklad – trénink Transformeru na TPU v4

```
 import flax.linen as nn import optax from flax.training import train_state
Jednoduchý Transformer blok (zkrácený)

class SimpleTransformer(nn.Module): d_model: int = 512 n_head: int = 8
n_layer: int = 6

@nn.compact
def __call__(self, x):
    for _ in range(self.n_layer):
```

```
        x = nn.SelfAttention(num_heads=self.n_head,
                             qkv_features=self.d_model)(x)
        x = nn.Dense(self.d_model)(x)
    return x

Inicializace

rng = jax.random.PRNGKey(0) model = SimpleTransformer() params =
model.init(rng, jnp.ones((1, 128, 512)))['params']
Optimizer (AdamW)

tx = optax.adamw(1e-4) state =
train_state.TrainState.create(apply_fn=model.apply, params=params, tx=tx)
JIT-ovaná tréninková smyčka

@jax.jit def train_step(state, batch): def loss_fn(params): logits =
state.apply_fn({'params': params}, batch['inputs']) loss =
optax.softmax_cross_entropy(logits, batch['targets']).mean() return loss
grads = jax.grad(loss_fn)(state.params) return
state.apply_gradients(grads=grads)
Příklad batchu (předpokládá se, že je již shardován na TPU)

batch = {'inputs': jnp.ones((8, 128, 512)), # 8 devices 'targets':
jnp.ones((8, 128, 512))} state = train_step(state, batch) print("Tréninková
ztráta:", state.loss)
```

Ekosystém kolem JAX

Knihovna	Popis	GitHub
Flax	Modulární NN knihovna (high-level)	https://github.com/google/flax
Haiku	DeepMind-inspirovaná NN knihovna (object-oriented)	https://github.com/deepmind/dm-haiku
Optax	Sada optimalizačních algoritmů a schedulérů	https://github.com/google-deepmind/optax
Equinox	„PyTorch-like" API s podporou JAX, zaměřená na jednoduchost	https://github.com/patrick-kidger/equinox
Chex	Testovací a debugging nástroje pro JAX kód	https://github.com/google/chex
JAX-MD	Simulace molekulární dynamiky a fyzikální modely	https://github.com/google/jax-md

Porovnání JAX vs. PyTorch vs. TensorFlow

Kritérium	JAX	PyTorch	TensorFlow
Výkon (XLA)	Vysoký - kompilace do XLA (GPU/TPU)	Dobrá - TorchScript, ale méně optimalizované pro TPU	Dobrá - TF-XLA, ale složitější API
Autodiff	Reverse-mode, forward-mode, vmap, pmap, grad	Autograd, torch.autograd, torch.jit	tf.GradientTape, tf.function
Funkcionální styl	Povinný (čisté funkce) - usnadňuje transformace	Volitelný (imperativní)	Volitelný (imperativní + graph)
Distribuce	pmap, pjit, mesh sharding - nativní podpora	torch.distributed, torch.nn.parallel	tf.distribute.Strategy
Ekosystém	Flax, Haiku, Optax, Equinox - rychle rostoucí	torchvision, torchtext, Lightning	Keras, tf.keras, TFLite
Kompatibilita s TPU	Native (XLA backend) - nejlepší volba	Experimentální (via torch_xla)	Native (TF-XLA)
Learning curve	Střední - nutnost pochopit funkcionální paradigm	Nízká - podobné NumPy/Python	Střední - graph vs. eager

Tipy a best practices

```
  Používejte jit – i pro malé funkce, aby se XLA optimalizoval.
  Vektorizujte s vmap – nahraďte for-loops, získáte SIMD-výhody.
  Rozdělujte data s pmap – pokud máte více zařízení, trénujte paralelně.
  Explicitní typy – při práci s TPU specifikujte jnp.bfloat16 nebo
jnp.float8 pro vyšší propustnost.
  Kontrola paměti – jax.debug.print a jax.profiler pomáhají najít memory
leaks.
  Reproducibilita – vždy inicializujte PRNGKey a předávejte jej explicitně.
  Kompatibilita s numpy – jax.numpy je drop-in, ale některé funkce (např.
np.linalg.eig) nejsou podporovány – použijte ekvivalenty z jax.scipy.
```

Budoucnost JAX

```
  JAX 3.0+ – podpora global device arrays (GDA) a asynchronní sharding.
```

```
  Sparse & Structured Matrices – rozšířená podpora pro sparse operace,
klíčové pro LLM.
  Compiler-level optimizations – další vylepšení XLA (např. fusion, tiling)
a podpora float8 na GPU/TPU.
  Interoperabilita s PyTorch – projekty jako torch2jax a jax2torch usnadní
migraci kódu.
  Rozšířený ekosystém – nové knihovny pro probabilistické programování
(např. NumPyro) a diferenciální fyziku (např. Diffrax).
```

== Závěr == JAX představuje moderní, výkonný a flexibilní nástroj pro výzkumníky i inženýry, kteří potřebují rychlé diferenciální výpočty a škálovatelnost napříč CPU, GPU i TPU. Díky čistému funkcionálnímu přístupu, silnému ekosystému (Flax, Optax, Haiku) a nativní podpoře pro XLA je JAX dnes jedním z hlavních pilířů vývoje velkých modelů (transformery, diffusion, reinforcement learning). Pokud chcete maximalizovat výkon na TPU, JAX + Flax je momentálně nejefektivnější cesta.

Odkazy a literatura

```
  [[https://github.com/google/jax|JAX – GitHub repository]]
  [[https://jax.readthedocs.io|JAX Documentation]]
  [[https://flax.readthedocs.io|Flax – Neural network library]]
  [[https://optax.readthedocs.io|Optax – Gradient processing & optimizers]]
  [[https://github.com/google/chex|Chex – Testing utilities for JAX]]
  [[https://arxiv.org/abs/1910.01408|“JAX: composable transformations of
Python+NumPy programs” – arXiv 2019]]
  [[https://cloud.google.com/tpu/docs/jax-quickstart|Google Cloud – JAX on
TPU Quickstart]]
  [[https://github.com/deepmind/dm-haiku|Haiku – DeepMind’s neural network
library]]
  [[https://github.com/google/jax-md|JAX-MD – Molecular dynamics library]]
```

From:
https://serviceit.cz/ - **IT ENCYKLOPEDIE**

Permanent link:
https://serviceit.cz/doku.php?id=jax

Last update: **2026/01/02 21:07**