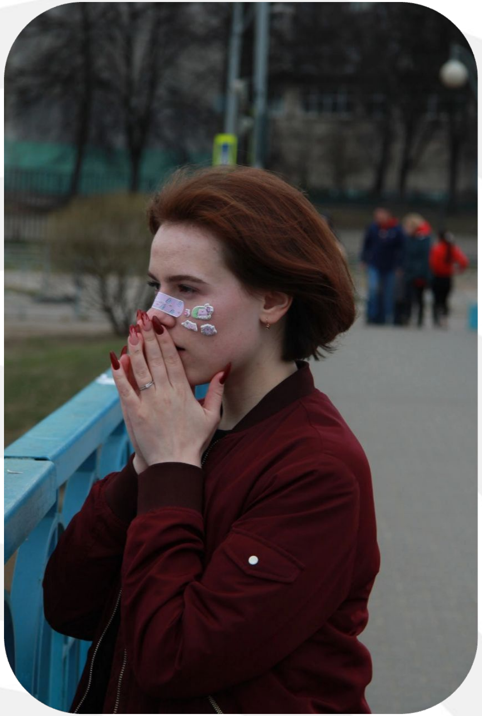
NumPy и PyTorch устарели?

Петренко Валерия

- Data scientist команды “Генеративный дизайн и языковые модели” в Ozone
- Студентка МФТИ
- @youngblackwitch



План доклада

01 Сравнение API Jax и Numpy

02 Сравнение API Jax и Pytorch

03 Сравнение производительности и потребления ресурсов

04 Экосистема Jax

05 Сравнение сообществ и применимости библиотек

01 Jax vs NumPy


NYP
NumPy

Array creation

```
array_1 = jnp.array([5, 7, 9])
print(isinstance(array_1, jax.Array)) # True
array_1 # Array([5, 7, 9], dtype=int32)
```

```
array_2 = jnp.arange(5)
print(isinstance(array_2, jax.Array)) # True
array_2 # Array([0, 1, 2, 3, 4], dtype=int32)
```

Array creation

```
jax_array = jnp.arange(10)
jax_array[0] = 10
```

Array creation

```
jax_array = jnp.arange(10)
jax_array[0] = 10
TypeError: '' object does
not support item
assignment. JAX arrays are
immutable
```

Array creation

```python
jax_array = jnp.arange(10)
jax_array[0] = 10
# TypeError: '' object does not support item assignment. JAX arrays are immutable
```

```python
array_1 = jnp.arange(9)
array_2 = array_1.at[0].set(9)
print(array_1)
# [0 1 2 3 4 5 6 7 8]
print(array_2)
# 10 1 2 3 4 5 6 7 8]
```

Array manipulation

```
jax_array = jnp.array([[1, 2, 3], [4, 5, 6]])

reshaped_array = jnp.reshape(jax_array, (3, 2))
rollaxis_array = jnp.rollaxis(jax_array, 1, 0)
jax_array_2 = jax_array[:, :2]

jax_array = np.array([[1], [2], [3], [4]])
squeezed_array = jnp.squeeze(jax_array)
```

Array manipulation

```
jax_array = jnp.array([[1, 2, 3], [4, 5, 6]])

reshaped_array = jnp.reshape(jax_array, (3, 2))
rollaxis_array = jnp.rollaxis(jax_array, 1, 0)
jax_array_2 = jax_array[:, :2]

jax_array = np.array([[1], [2], [3], [4]])
squeezed_array = jnp.squeeze(jax_array)
```

Array manipulation

```
jax_array = jnp.array([[1, 2, 3], [4, 5, 6]])

reshaped_array = jnp.reshape(jax_array, (3, 2))
rollaxis_array = jnp.rollaxis(jax_array, 1, 0)
jax_array_2 = jax_array[:, :2]

jax_array = np.array([[1], [2], [3], [4]])
squeezed_array = jnp.squeeze(jax_array)
```

Array manipulation

```
jax_array = jnp.array([[1, 2, 3], [4, 5, 6]])

reshaped_array = jnp.reshape(jax_array, (3, 2))
rollaxis_array = jnp.rollaxis(jax_array, 1, 0)
jax_array_2 = jax_array[:, :2]

jax_array = np.array([[1], [2], [3], [4]])
squeezed_array = jnp.squeeze(jax_array)
```

Matrix operation

```
jax_array_1 = jnp.array([1.0, 2.0, 3.0])
jax_array_2 = jnp.array([4.0, 5.0, 6.0])

array_sum = jax_array_1 + jax_array_2
# array([5., 7., 9.])
array_product = jax_array_1 * jax_array_2
# array([ 4., 10., 18.])
exp_array_1 = np.exp(jax_array_1)
# array([ 2.71828183,  7.3890561 , 20.08553692])
```

Matrix operation

```
jax_array_1 = jnp.array([1.0, 2.0, 3.0])
jax_array_2 = jnp.array([4.0, 5.0, 6.0])

array_sum = jax_array_1 + jax_array_2
# array([5., 7., 9.])
array_product = jax_array_1 * jax_array_2
# array([ 4., 10., 18.])
exp_array_1 = np.exp(jax_array_1)
# array([ 2.71828183,  7.3890561 , 20.08553692])
```

Matrix operation

```
jax_array_1 = jnp.array([1.0, 2.0, 3.0])
jax_array_2 = jnp.array([4.0, 5.0, 6.0])

array_sum = jax_array_1 + jax_array_2
# array([5., 7., 9.])
array_product = jax_array_1 * jax_array_2
# array([ 4., 10., 18.])
exp_array_1 = np.exp(jax_array_1)
# array([ 2.71828183,  7.3890561 , 20.08553692])
```

ozon{ech

Linear algebra

```
jax_array = jnp.array([[2., 1.],[1., 2.]])
cholesky_jax = jnp.linalg.cholesky(jax_array)
# Array([[1.4142135 , 0.        ],
#        [0.70710677, 1.2247449 ]], dtype=float32)
eigvals_jax = jnp.linalg.eigvals(jax_array)
# Array([3.+0.j, 1.+0.j], dtype=complex64)
jax_array_1 = jnp.array([[1, 2], [3, 5]])
jax_array_2 = jnp.array([1, 2])
result = jnp.linalg.solve(jax_array_1, jax_array_2)
# Array([-1.0000002,  1.0000001], dtype=float32)
```

Linear algebra

```
jax_array = jnp.array([[2., 1.],[1., 2.]])
cholesky_jax = jnp.linalg.cholesky(jax_array)
# Array([[1.4142135 , 0.        ],
#        [0.70710677, 1.2247449 ]], dtype=float32)
eigvals_jax = jnp.linalg.eigvals(jax_array)
# Array([3.+0.j, 1.+0.j], dtype=complex64)
jax_array_1 = jnp.array([[1, 2], [3, 5]])
jax_array_2 = jnp.array([1, 2])
result = jnp.linalg.solve(jax_array_1, jax_array_2)
# Array([-1.0000002,  1.0000001], dtype=float32)
```

Linear algebra

```
jax_array = jnp.array([[2., 1.],[1., 2.]])
cholesky_jax = jnp.linalg.cholesky(jax_array)
# Array([[1.4142135 , 0.        ],
#        [0.70710677, 1.2247449 ]], dtype=float32)
eigvals_jax = jnp.linalg.eigvals(jax_array)
# Array([3.+0.j, 1.+0.j], dtype=complex64)
jax_array_1 = jnp.array([[1, 2], [3, 5]])
jax_array_2 = jnp.array([1, 2])
result = jnp.linalg.solve(jax_array_1, jax_array_2)
# Array([-1.0000002,  1.0000001], dtype=float32)
```

Jax vs NumPy: Pseudorandom numbers

Pseudorandom numbers NumPy

```
np.random.seed(0)
prn_1 = np.random.uniform(size=1)
# [0.38344152]
prn_2 = np.random.uniform(size=1)
# [0.79172504]
```

Pseudorandom numbers NumPy

```
np.random.seed(0)
prn_1 = np.random.uniform(size=1)
# [0.38344152]
prn_2 = np.random.uniform(size=1)
# [0.79172504]
```

Pseudorandom numbers NumPy

```
np.random.seed(0)
individ_gen = [np.random.uniform() for _ in range(3)]
print("individually:", np.stack(individ_gen))
# individually: [0.5488135  0.71518937 0.60276338]
np.random.seed(0)
print("all at once: ", np.random.uniform(size=3))
# all at once:  [0.5488135  0.71518937 0.60276338]
```

Pseudorandom numbers Jax

```
np.random.seed(0)

def bar(): return np.random.uniform()
def baz(): return np.random.uniform()

def foo(): return bar() + 2 * baz()
print(foo()) # 1.9791922366721637
```

ozon{ech

Pseudorandom numbers Jax

```
np.random.seed(0)

def bar(): return np.random.uniform()
def baz(): return np.random.uniform()

def foo(): return bar() + 2 * baz()
print(foo()) # 1.9791922366721637
```

В Jax так нельзя!

Pseudorandom numbers Jax

В JAX генерация PRNG:

- Воспроизводима
- Параллелизуема
- Векторизуема

Функциональный подход

Если функция удовлетворяет следующим условиям, она считается чистой:

- Все входные данные поступают из параметров

Функциональный подход

Если функция удовлетворяет следующим условиям, она считается чистой:

- Все входные данные поступают из параметров
- Все выходные данные возвращаются из функции

Функциональный подход

Если функция удовлетворяет следующим условиям, она считается чистой:

- Все входные данные поступают из параметров
- Все выходные данные возвращаются из функции
- При отправке одних и тех же входных данных результаты всегда должны быть одинаковыми

Функциональный подход

```
class StatefulClass

  state: State
  def stateful_method(*args, **kwargs) -> Output:
```

Функциональный подход

```
class StatelessClass

  def stateless_method(state: State, *args, **kwargs)
-> (Output, State):
```

Pseudorandom numbers Jax

```
key = jax.random.key(42)
print(key)
# Array((), dtype=key<fry>)
# overlaying: [ 0 42]
```

Pseudorandom numbers Jax

```
key = jax.random.key(42)
print(key)
# Array((), dtype=key<fry>)
# overlaying: [ 0 42]
```

```
print(jax.random.normal(key))
# -0.18471177
print(jax.random.normal(key))
# -0.18471177
```

ozon tech

Pseudorandom numbers Jax

```
key = jax.random.PRNGKey(42)

random_numbers = jax.random.uniform(key, shape=(3,))
# [0.57414436 0.10015821 0.05946112]

key, subkey = jax.random.split(key)

new_random_numbers = jax.random.uniform(subkey, shape=(3,))
# [0.2851702 0.6119449 0.1756525]
```

ozon{ech

Pseudorandom numbers Jax

```
key = jax.random.key(42)
subkeys = jax.random.split(key, 3)
sequence = np.stack([jax.random.normal(subkey) for subkey
in subkeys])
print("individually:", sequence)
# individually: [-0.04838832  0.10796154 -1.2226542]

key = jax.random.key(42)
print("all at once: ", jax.random.normal(key, shape=(3,)))
# all at once:  [ 0.18693547 -1.2806505  -1.5593132]
```

Pseudorandom numbers Jax

```
vectorized_seq = jax.vmap(random.normal)(subkeys)
print("vectorized:", vectorized_seq)
# vectorized: [-0.04838832  0.10796154 -1.2226542 ]
```


I'M YOU BUT STRONGER
WHO ARE YOU?
NumPy

Jax vs NumPy: JIT-компиляция

Что такое JIT?

«**JIT-компилятор** – это магия. Вы просто пишете свой код, и ваша программа начинает работать быстрее»

Джон Розенберг, создатель HotSpot JIT-компилятора для Java Virtual Machine

Интерпретатор в Python

Токенизация

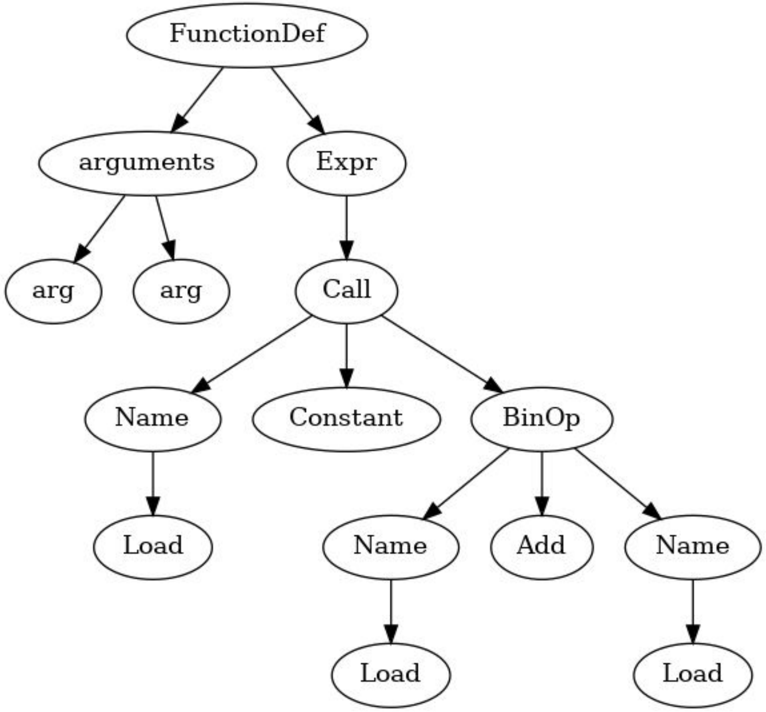
Интерпретатор в Python

Токенизация

Парсер (PEG)

Интерпретатор в Python

- Токенизация
- Парсер (PEG)
- **AST и оптимизация**



Интерпретатор в Python

- Токенизация
- Парсер (PEG)
- AST и оптимизация
- **Компиляция байт-кода**

Интерпретатор в Python

- Токенизация
- Парсер (PEG)
- AST и оптимизация
- Компиляция байт-кода
- **PVM**

ozon{ech

Интерпретатор в Python

- Токенизация
- Парсер (PEG)
- AST и оптимизация
- Компиляция байт-кода
- PVM
- **Выполнение байт-кода**

ozon{ech

Интерпретатор в Python

- Токенизация
- Парсер (PEG)
- AST и оптимизация
- Компиляция байт-кода
- PVM
- Выполнение байт-кода

Что такое JIT в Jax?

Создание вычислительного графа

OpenXLA

Что такое JIT в Jax?

Создание вычислительного графа

Оптимизация в XLA

OpenXLA

Что такое JIT в Jax?

- Создание вычислительного графа
- Оптимизация в XLA
- Компиляция машинного кода в XLA

OpenXLA

Что такое JIT в Jax?

- Создание вычислительного графа
- Оптимизация в XLA
- Компиляция машинного кода в XLA
- Выполнение машинного кода

Jaxprs

```
def f(arg1, arg2):
    temp = jnp.sin(arg2)
    temp = temp * 3
    temp = temp + arg1
    return jnp.sum(temp)

arg_1 = jnp.zeros(8)
arg_2 = jnp.ones(8)
make_jaxpr(f)(arg_1, arg_2)
```

Jaxprs

```python
def f(arg1, arg2):
    temp = jnp.sin(arg2)
    temp = temp * 3
    temp = temp + arg1
    return jnp.sum(temp)

arg_1 = jnp.zeros(8)
arg_2 = jnp.ones(8)
make_jaxpr(f)(arg_1, arg_2)
```

Var* **Var+**

```
{ 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,) }
```

Eqn*

Expr*

ozon{ech

Accelerated Linear Algebra

- Fusion

OpenXLA

Accelerated Linear Algebra

- Fusion
- Constant Folding

OpenXLA

Accelerated Linear Algebra

- Fusion
- Constant Folding
- Dead Code Elimination

OpenXLA

Accelerated Linear Algebra

- Fusion
- Constant Folding
- Dead Code Elimination
- Common Subexpression Elimination

OpenXLA

Accelerated Linear Algebra

- Fusion
- Constant Folding
- Dead Code Elimination
- Common Subexpression Elimination
- Algebraic Simplifications

OpenXLA

Accelerated Linear Algebra

- Fusion
- Constant Folding
- Dead Code Elimination
- Common Subexpression Elimination
- Algebraic Simplifications
- Layout Optimization

OpenXLA

JIT-компиляция NumPy

```python
def myfunc(x):
    return np.dot(x, x)

x = np.array([1.0,2.0,3.0])
```

```python
from numba import jit

@jit(nopython=True)
def myfunc(x):
    return np.dot(x, x)

x = np.array([1.0, 2.0, 3.0])
```

JIT-компиляция Jax

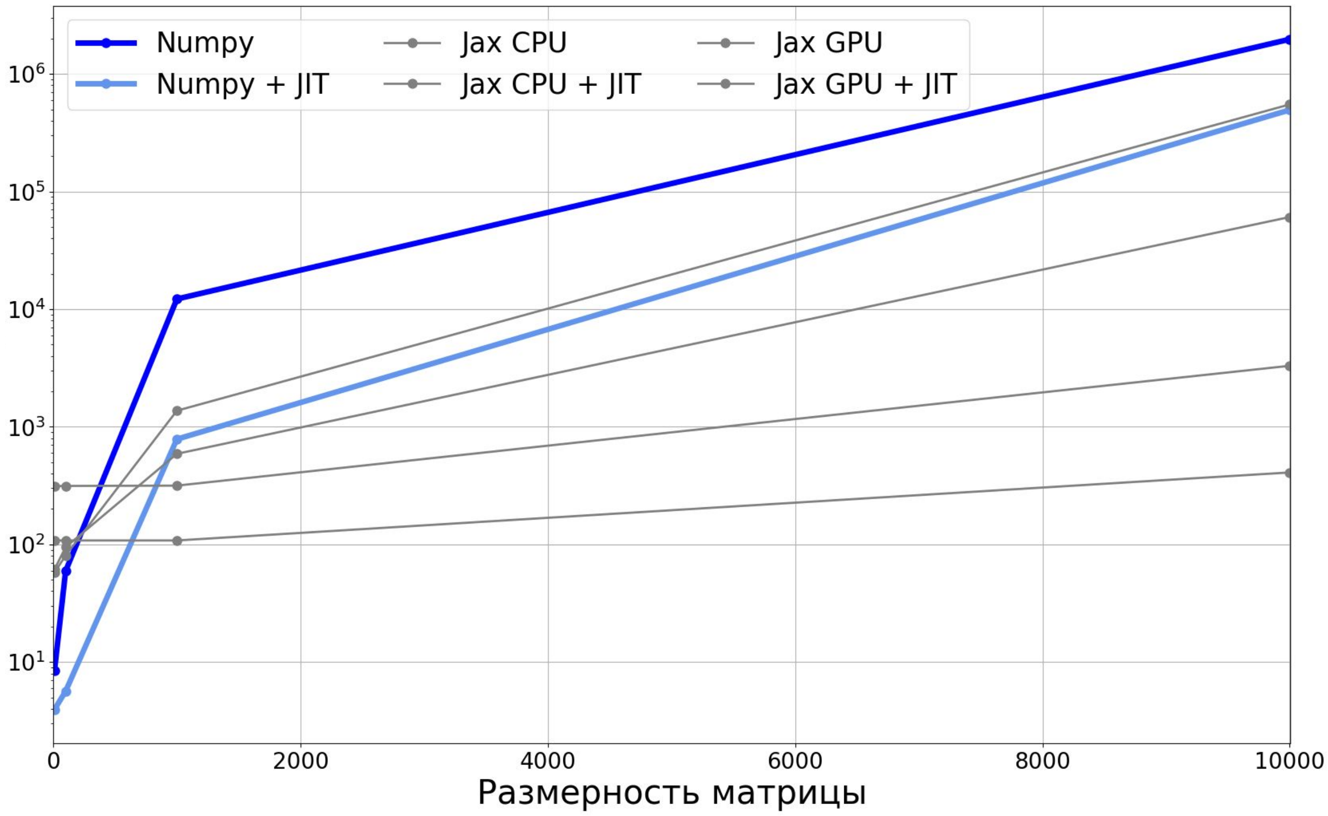
```
def myfunc(x):
    return jnp.dot(x, x)

jit_myfunc = jax.jit(myfunc)

x = jnp.array([1.0, 2.0, 3.0])
```

Performance benchmarks: Numpy vs Jax

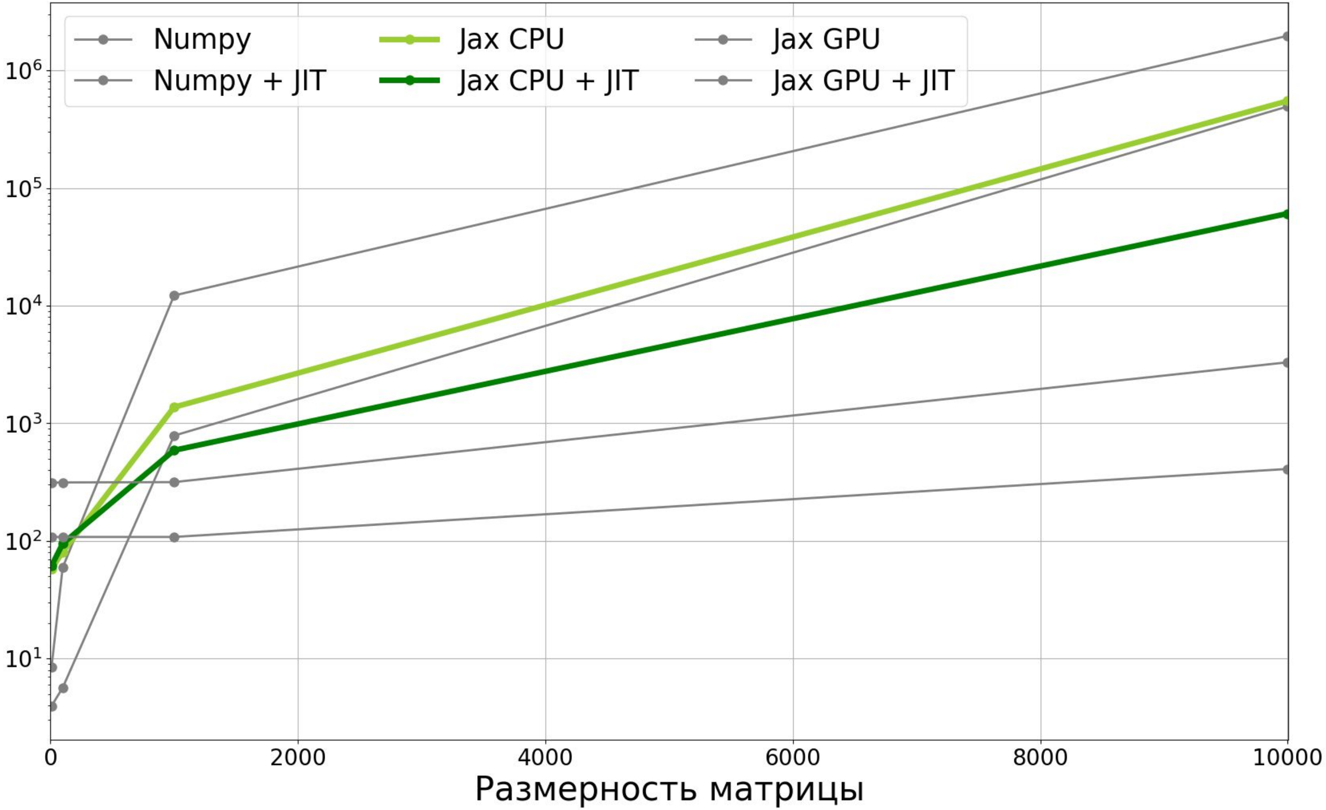
```python
def f(x):
  return -4*x*x*x + 9*x*x + 6*x - 3
```

ozon tech

Performance benchmarks: Numpy vs Jax, µs

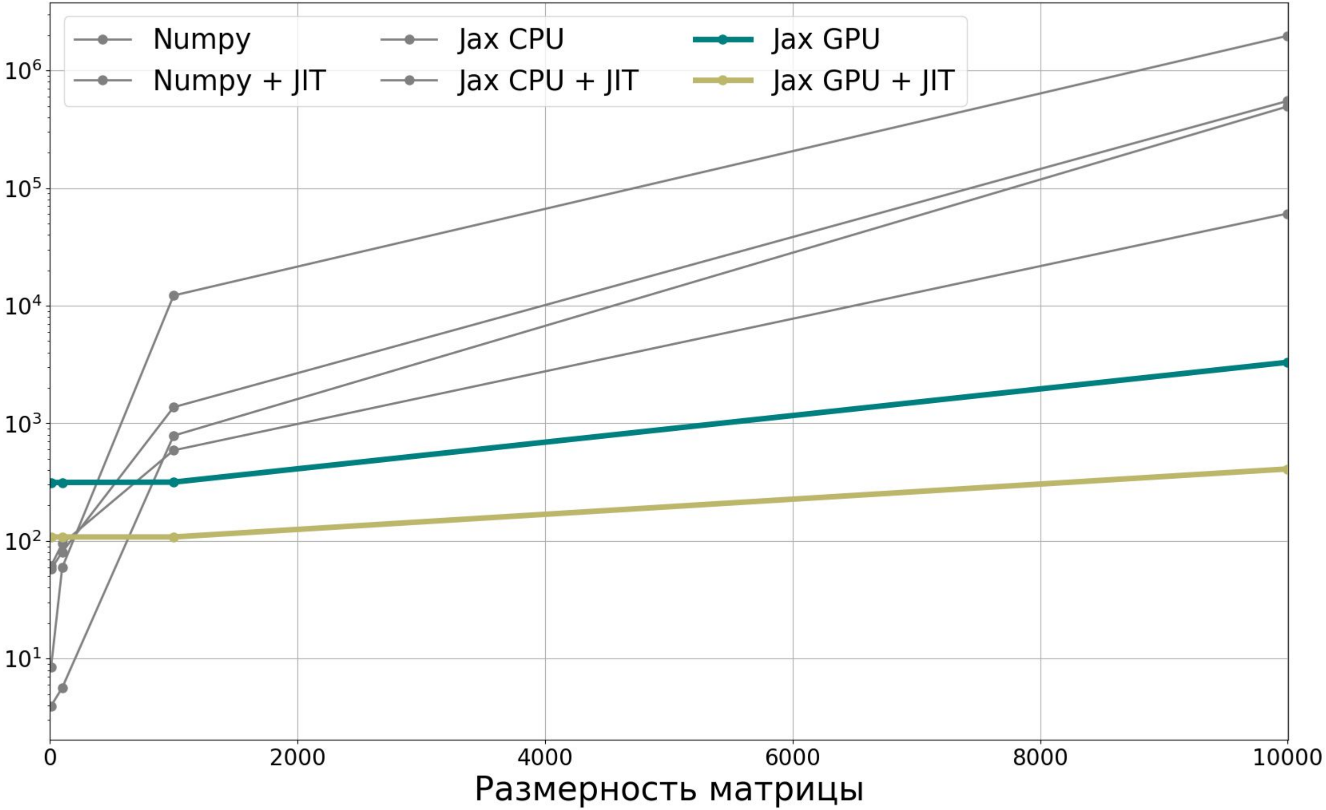


Performance benchmarks: Numpy vs Jax, µs



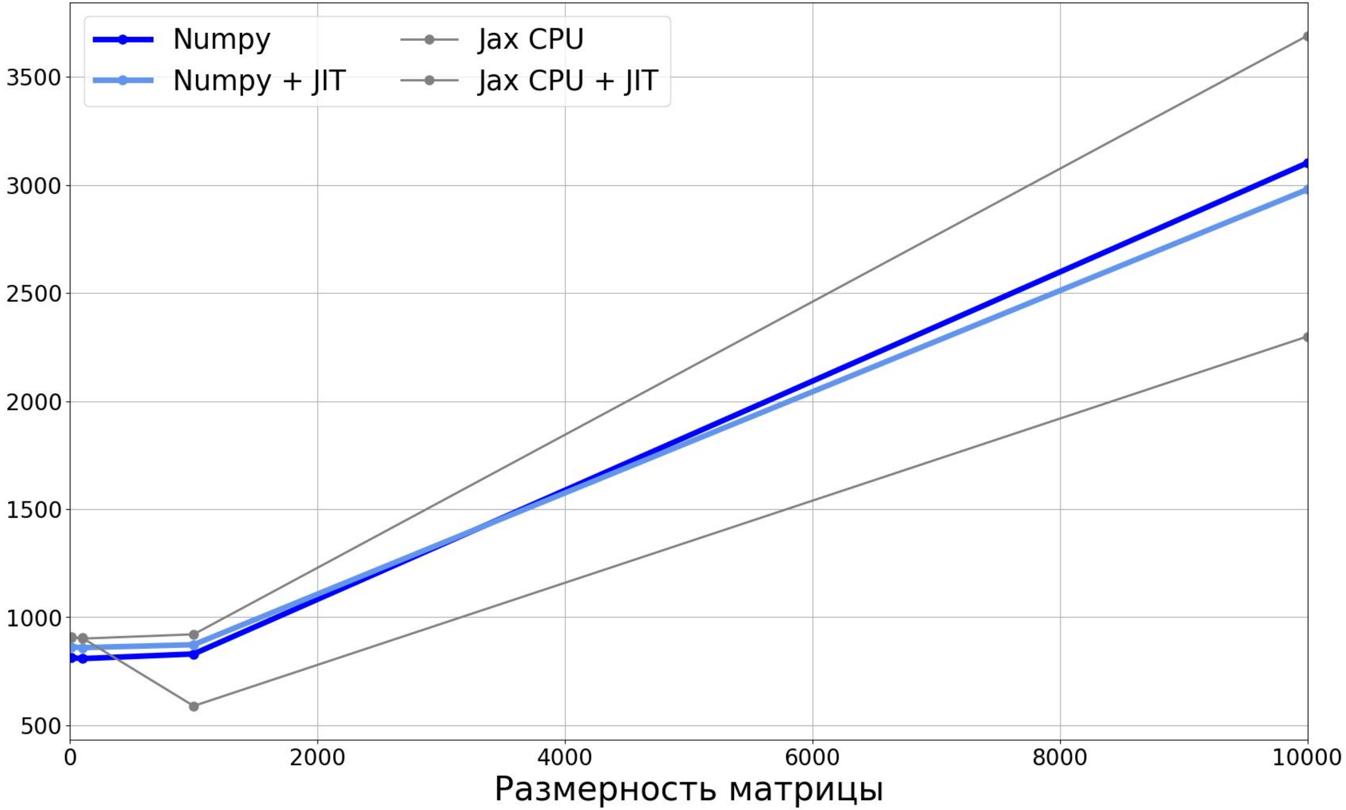
ozon{ech

Performance benchmarks: Numpy vs Jax, µs

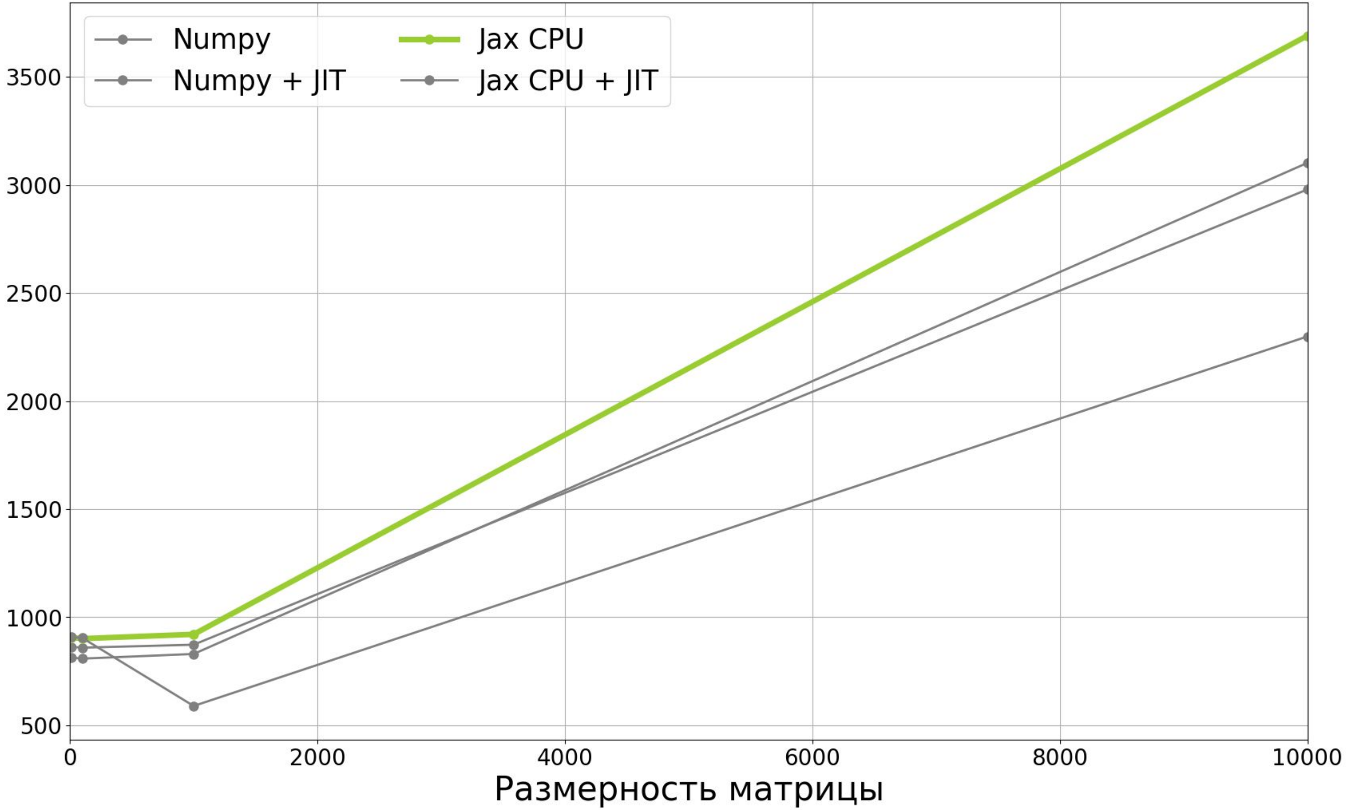


ozon{ech

Memory benchmarks: Numpy vs Jax, RAM MiB

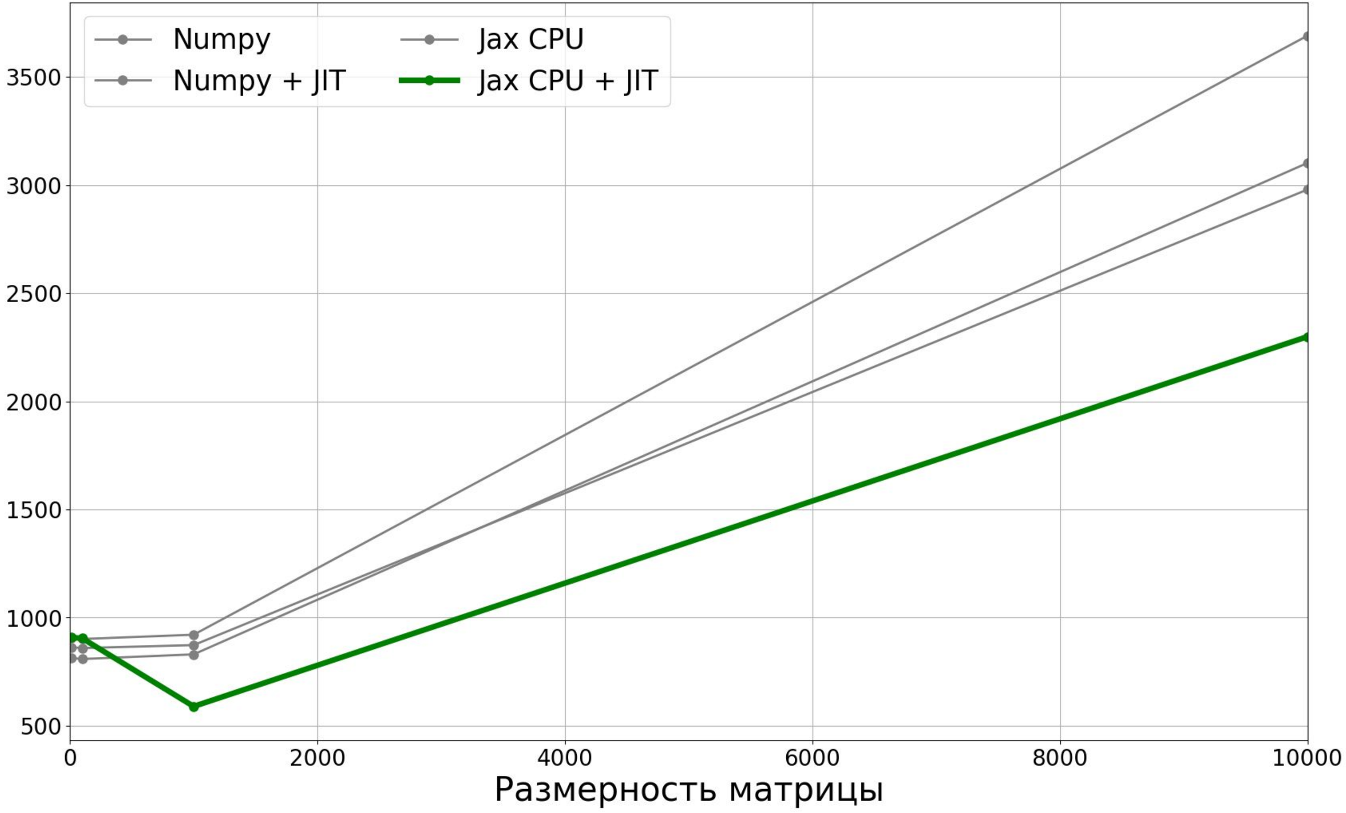


Memory benchmarks: Numpy vs Jax, RAM MiB



ozon{ech

Memory benchmarks: Numpy vs Jax, RAM MiB




JIT в
массы!

Jax vs NumPy: Automatic vectorization

Automatic vectorization NumPy

```python
def myfunc(a, b):
    "Return a-b if a>b, otherwise return a+b"
    if a > b:
        return a - b
    else:
        return a + b

vfunc = np.vectorize(myfunc, otypes=[float])
vfunc([1, 2, 3, 4], 2)
# array([3., 4., 1., 2.])
```

ozon{ech

Automatic vectorization NumPy

```python
def my_polyval(a, b):
    _a = list(a)
    res = _a.pop(0)
    while _a:
        res = res*b + _a.pop(0)
    return res

vec_polyval = np.vectorize(my_polyval, excluded=['a'])
vec_polyval.excluded.add(0)
vec_polyval(a=[1, 3, 5], b=[0, 1]) # array([5, 9])
```

ozon{ech

Automatic vectorization NumPy

```python
def my_polyval(a, b):
    _a = list(a)
    res = _a.pop(0)
    while _a:
        res = res*b + _a.pop(0)
    return res

vec_polyval = np.vectorize(my_polyval, excluded=['a'])
vec_polyval.excluded.add(0)
vec_polyval(a=[1, 3, 5], b=[0, 1]) # array([5, 9])
```

ozon{ech

Automatic vectorization Jax

```
def myfunc(a, b):
    "Return a-b if a>b, otherwise return a+b"
    return jnp.where(a > b, a - b, a + b)

vfunc = jax.vmap(myfunc, in_axes=(0, None))
array_1 = jnp.asarray([1, 2, 3, 4])
array_2 = jnp.asarray(2)
vfunc(array_1, array_2)
# Array([3, 4, 1, 2], dtype=int32)
```

Don’t use if-else!

Automatic vectorization Jax

```
def myfunc(a, b):
    "Return a-b if a>b, otherwise return a+b"
    return jnp.where(a > b, a - b, a + b)

vfunc = jax.vmap(myfunc, in_axes=(0, None))
array_1 = jnp.asarray([1, 2, 3, 4])
array_2 = jnp.asarray(2)
vfunc(array_1, array_2)
# Array([3, 4, 1, 2], dtype=int32)
```

Automatic vectorization Jax

```
def my_polyval(a, b):
    _a = list(a)
    res = _a.pop(0)
    while _a:
        res = res*b + _a.pop(0)
    return res

vec_polyval = jax.vmap(my_polyval, in_axes=(None, 0))
vec_polyval(jnp.asarray([1, 3, 5]), jnp.asarray([0, 1]))
# Array([5, 9], dtype=int32)
```

ozon{ech

Performance benchmarks: Numpy vs Jax

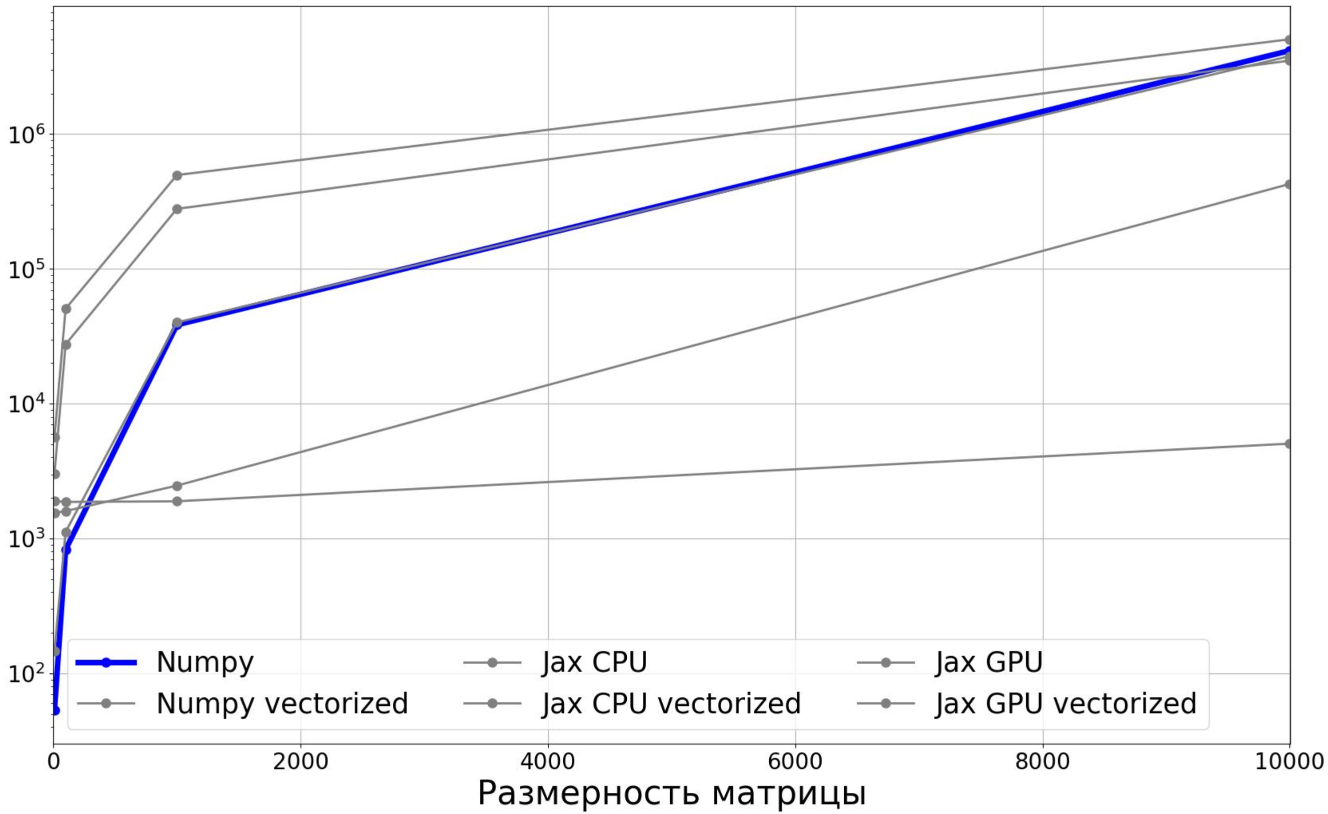
```
def calculate_distances(x, y):
    distances = []
    nrows, _ = x.shape
    for i in range(nrows):
         dist = (x[i] - y)**2
        distances_from_point = np.sqrt(dist.sum(axis=1))
        distances.append(distances_from_point)
    return np.array(distances)
```

Performance benchmarks: Numpy vs Jax

```
def calculate_distances_from_single_point(xi, y):
    return np.sqrt(((xi - y)**2).sum(axis=1))

vec_calculate_distances = np.vectorize(
calculate_distances_from_single_point,signature='(n),
(m,n)->(m)')

vmapped_calculate_distances = jax.vmap(
calculate_distances_from_single_point, in_axes=(0, None))
```

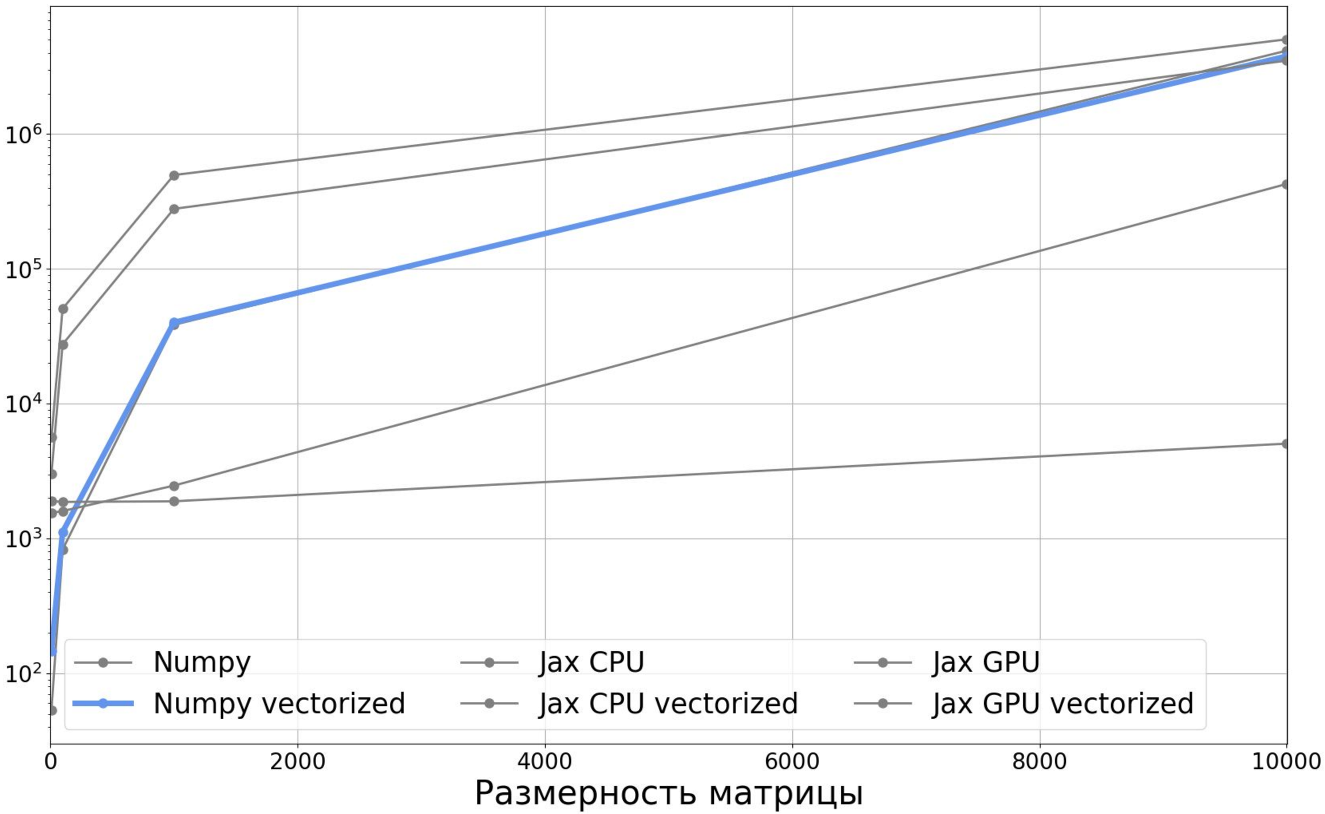
ozon{ech

Performance benchmarks: Numpy vs Jax, µs

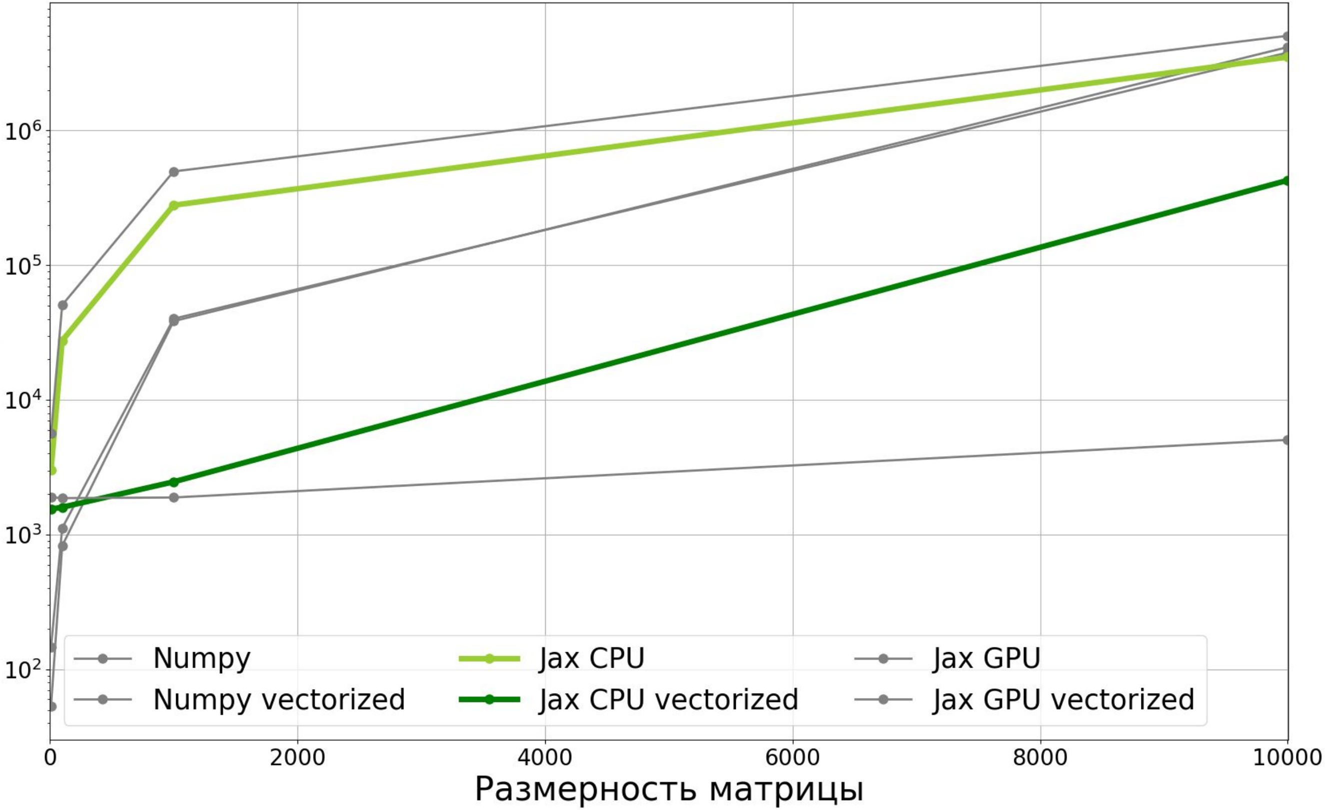


ozon{ech

Performance benchmarks: Numpy vs Jax, µs

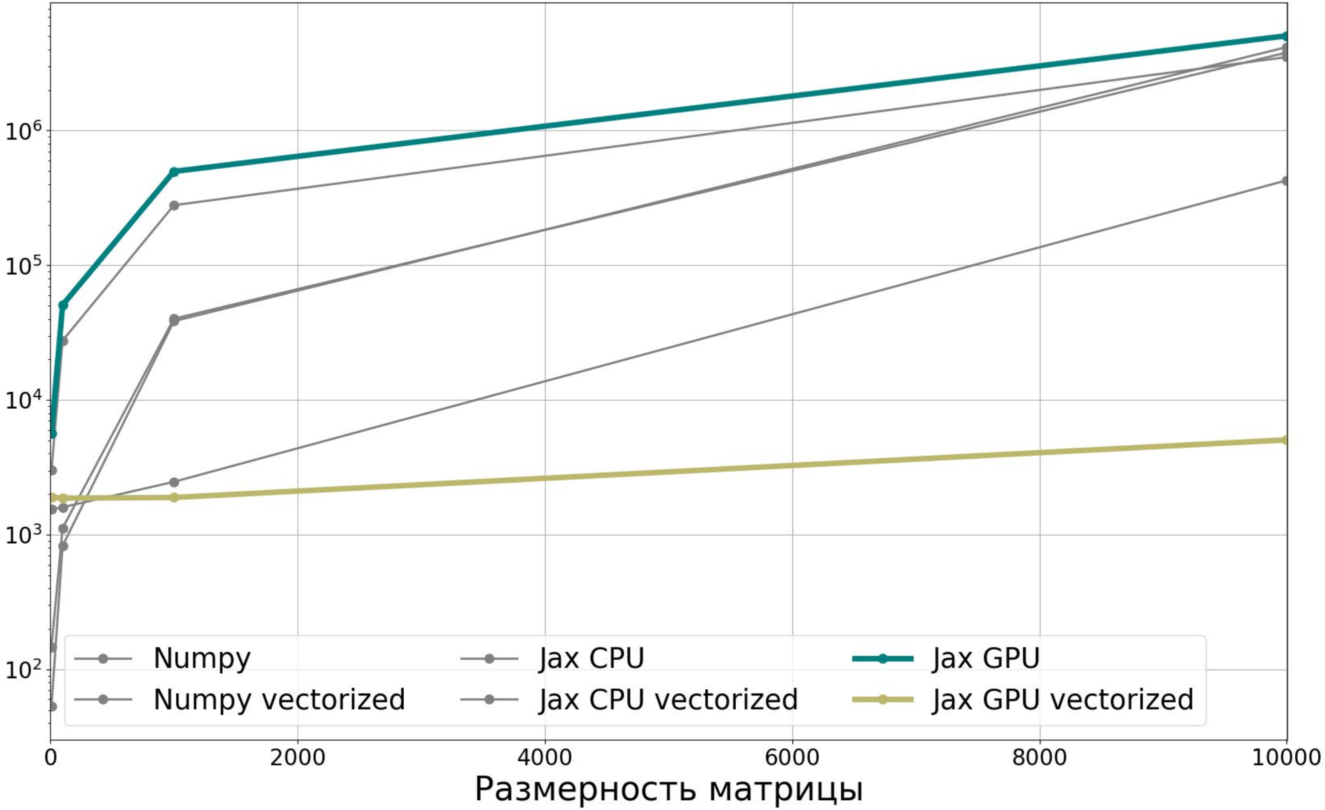


Performance benchmarks: Numpy vs Jax, μs



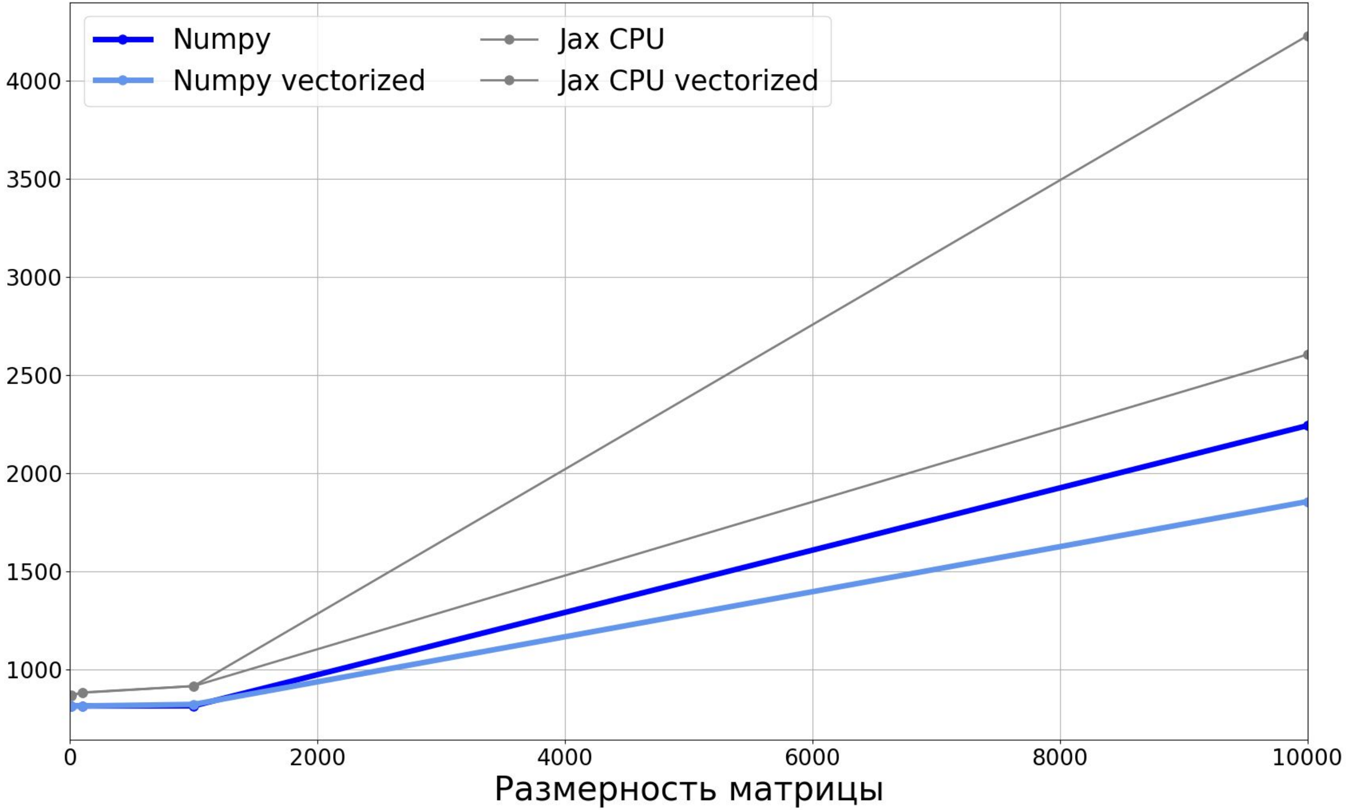
ozon{ech

Performance benchmarks: Numpy vs Jax, µs



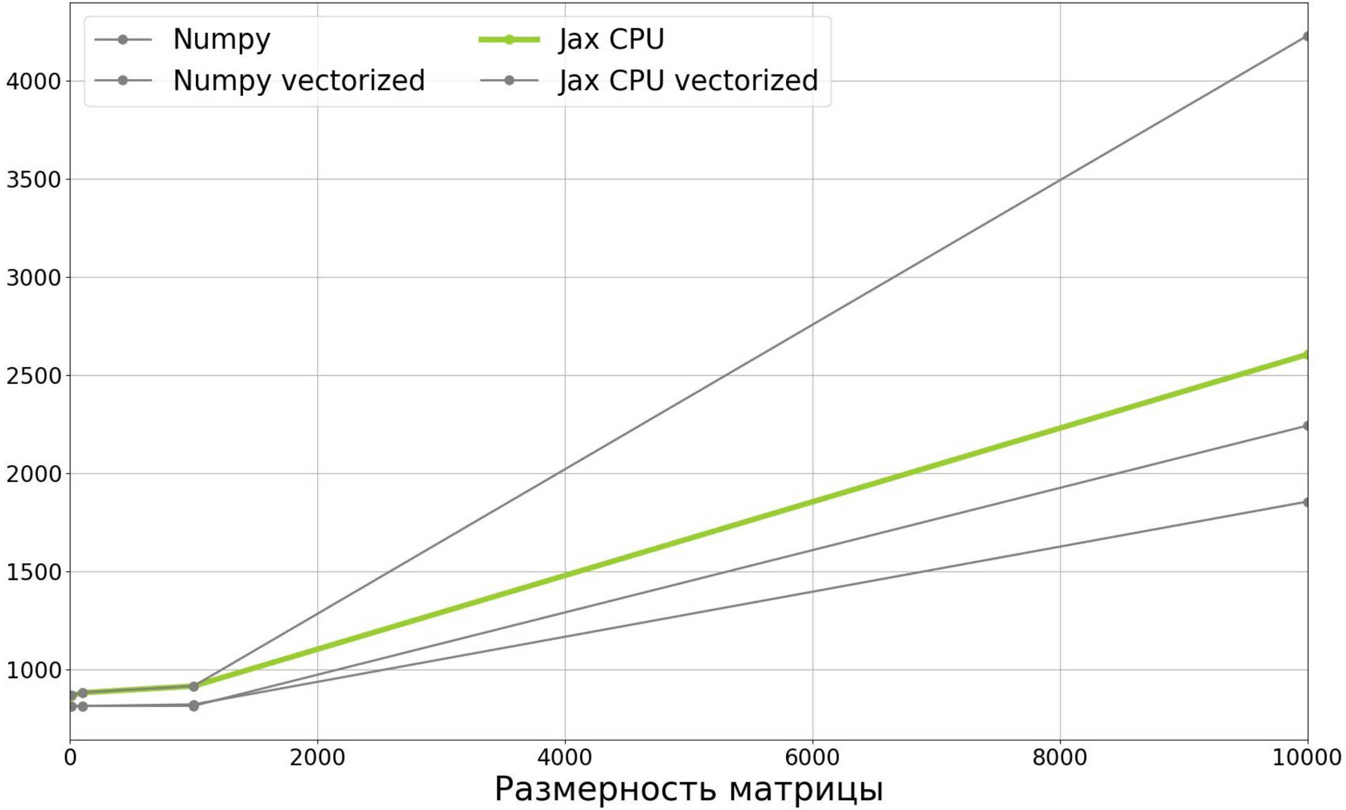
ozon{ech

Memory benchmarks: Numpy vs Jax, RAM MiB



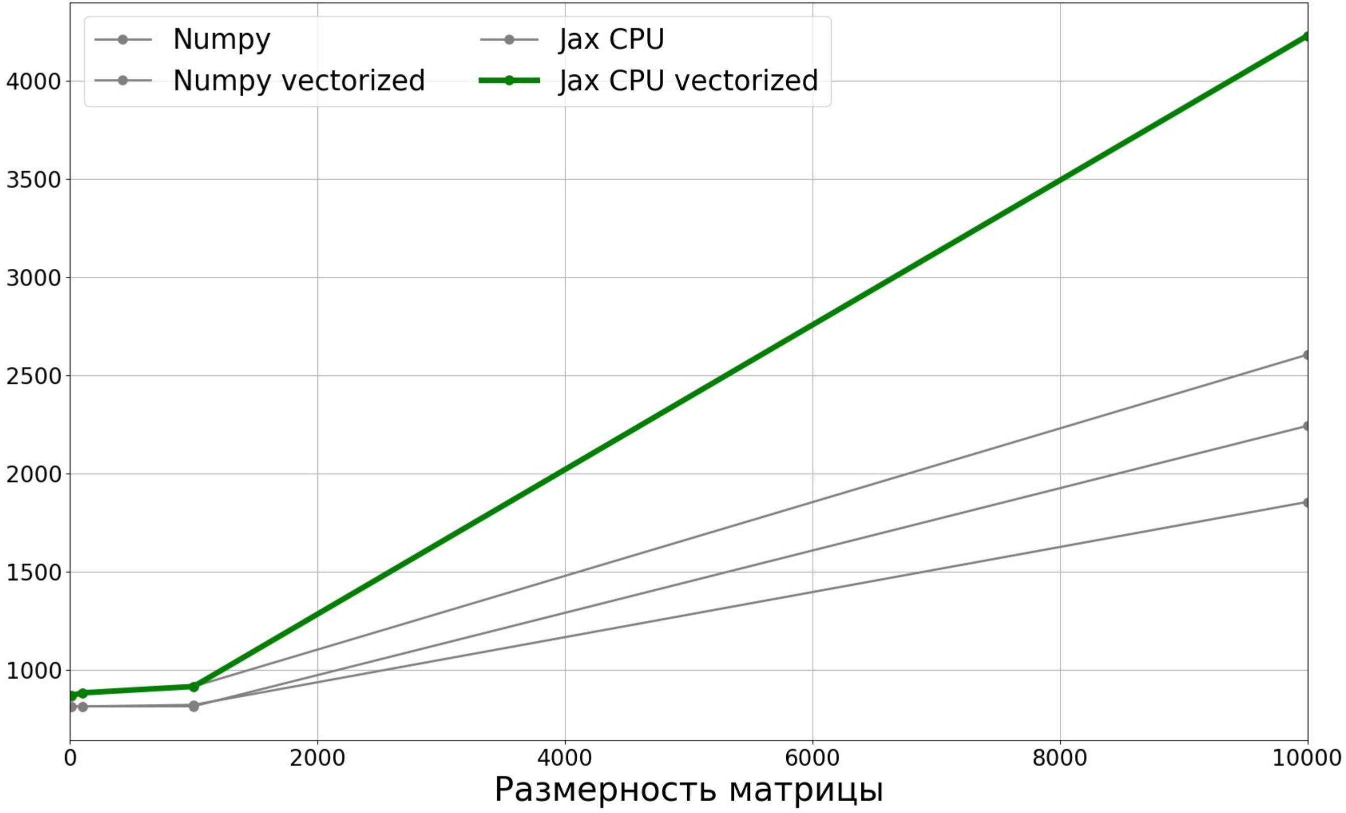
ozon{ech

Memory benchmarks: Numpy vs Jax, RAM MiB



ozon{ech

Memory benchmarks: Numpy vs Jax, RAM MiB



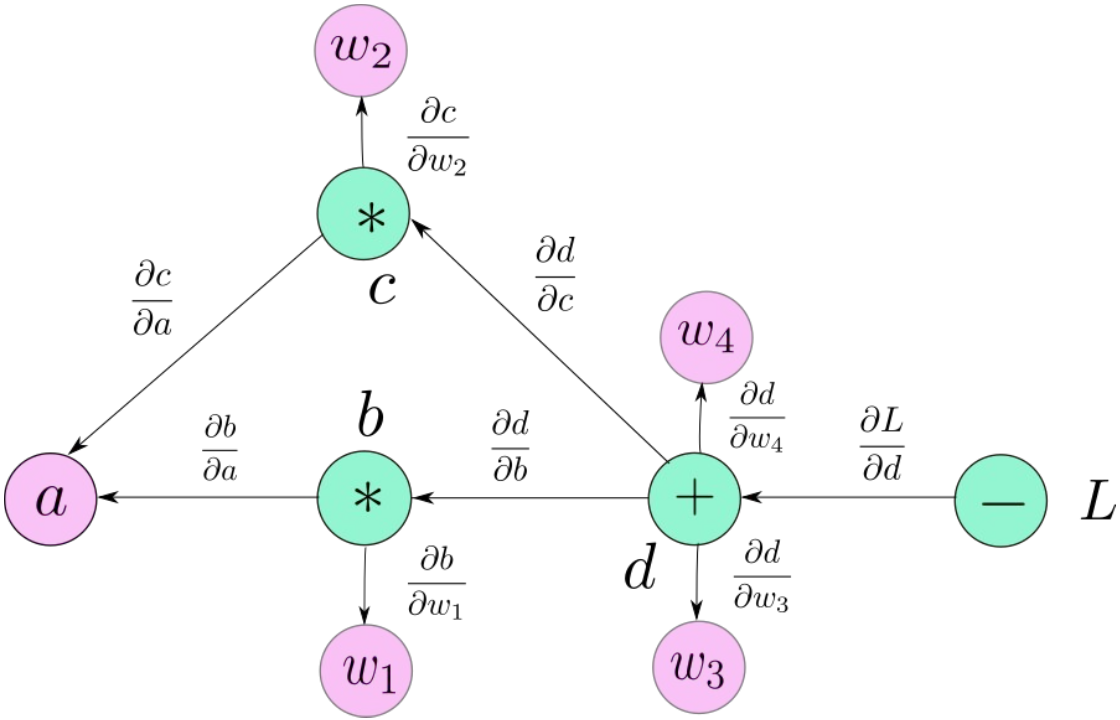
ozon{ech

02

Jax vs Pytorch

Jax vs PyTorch: Automatic differentiation

Automatic differentiation PyTorch



Automatic differentiation PyTorch

```
x = torch.tensor(2.0, requires_grad=True)

function = x**2 + 3 * x + 2
function.backward()
print(x.grad)
# tensor(7.)
print(function)
# tensor(12., grad_fn=<AddBackward0>)
```

Automatic differentiation PyTorch

```python
def my_func(x):
    if x > 2: return x**2
    else: return x + 2

x = torch.tensor(3.0, requires_grad=True)
function = my_func(x)
function.backward()
print(x.grad) # tensor(6.)
```

Automatic differentiation PyTorch

```
def my_func(x):
     return x**3

x = torch.tensor(2.0, requires_grad=True)
function = my_func(x)
grad1 = torch.autograd.grad(function, x,
create_graph=True)[0]
grad2 = torch.autograd.grad(grad1, x)[0]
print(grad2) # tensor(12.)
```

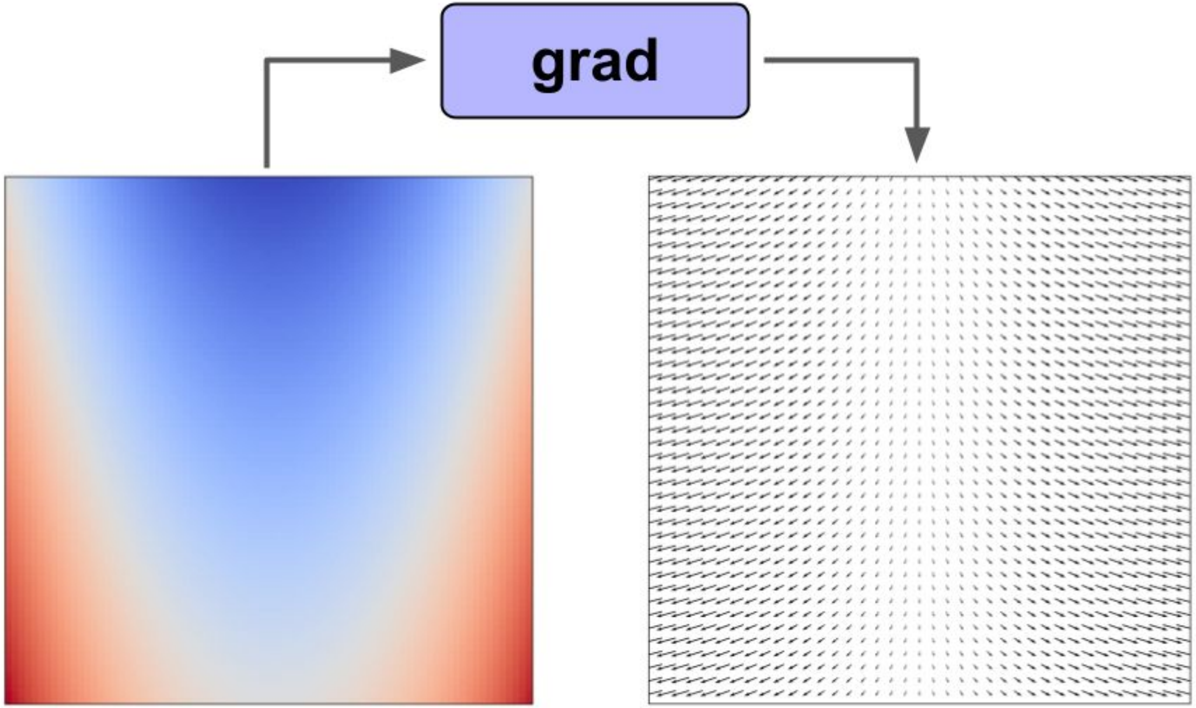
Automatic differentiation PyTorch

```python
def my_func(x):
      return x**3

x = torch.tensor(2.0, requires_grad=True
function = my_func(x)
grad1 = torch.autograd.grad(function, x,
create_graph=True)[0]
grad2 = torch.autograd.grad(grad1, x)[0]
print(grad2) # tensor(12.)
```

```python
@torch.jit.script
def my_func(x):
      return x**3
```

ozon{ech

Automatic differentiation Jax



Automatic differentiation Jax

```python
def func(x):
    return x**2 + 3 * x + 2

grad_func = jax.grad(func)
grad_value = grad_func(2.0)
print(grad_value) # 7.0
print(grad_func) # <function func at 0x30a8625c0>
```

Automatic differentiation Jax

```
def my_func(x):
    return jax.lax.cond(x > 2, lambda _: x**2,
                        lambda _: x + 2, operand=None)

grad_func = jax.grad(my_func)
grad_value = grad_func(3.0)
print(grad_value) # 6.0
```

Automatic differentiation Jax

```python
def func(x):
    return x**3

grad_func = jax.grad(func)
grad2_func = jax.grad(grad_func)
grad2_value = grad2_func(2.0)
print(grad2_value) # 12.0
```

ozon{ech

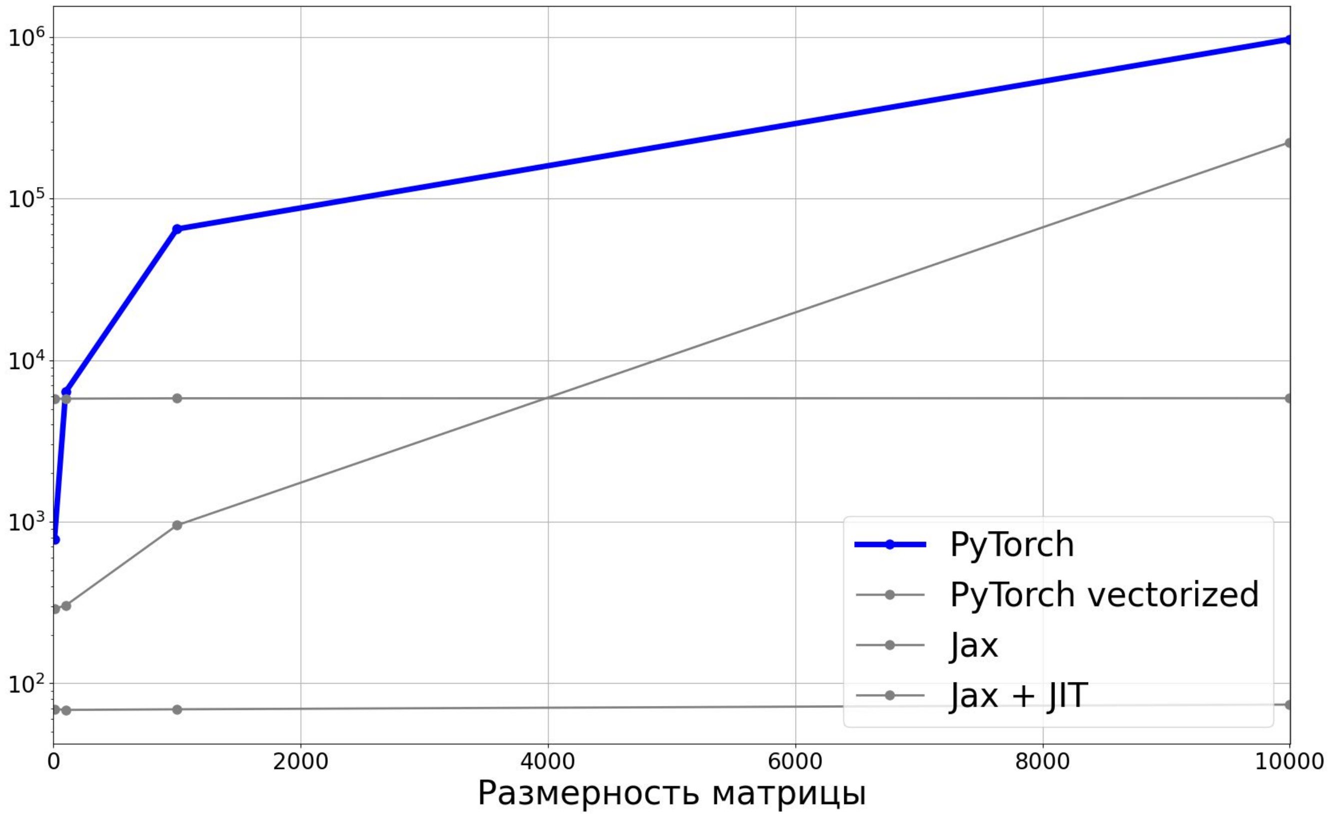
Automatic differentiation Jax

```python
def func(x):
    return x**3

grad_func = jax.grad(func)
grad2_func = jax.grad(grad_func)
grad_func = jax.jit(jax.grad(grad2_func))
grad2_value = grad2_func(2.0)
print(grad2_value) # 12.0
```

Performance benchmarks: PyTorch vs Jax

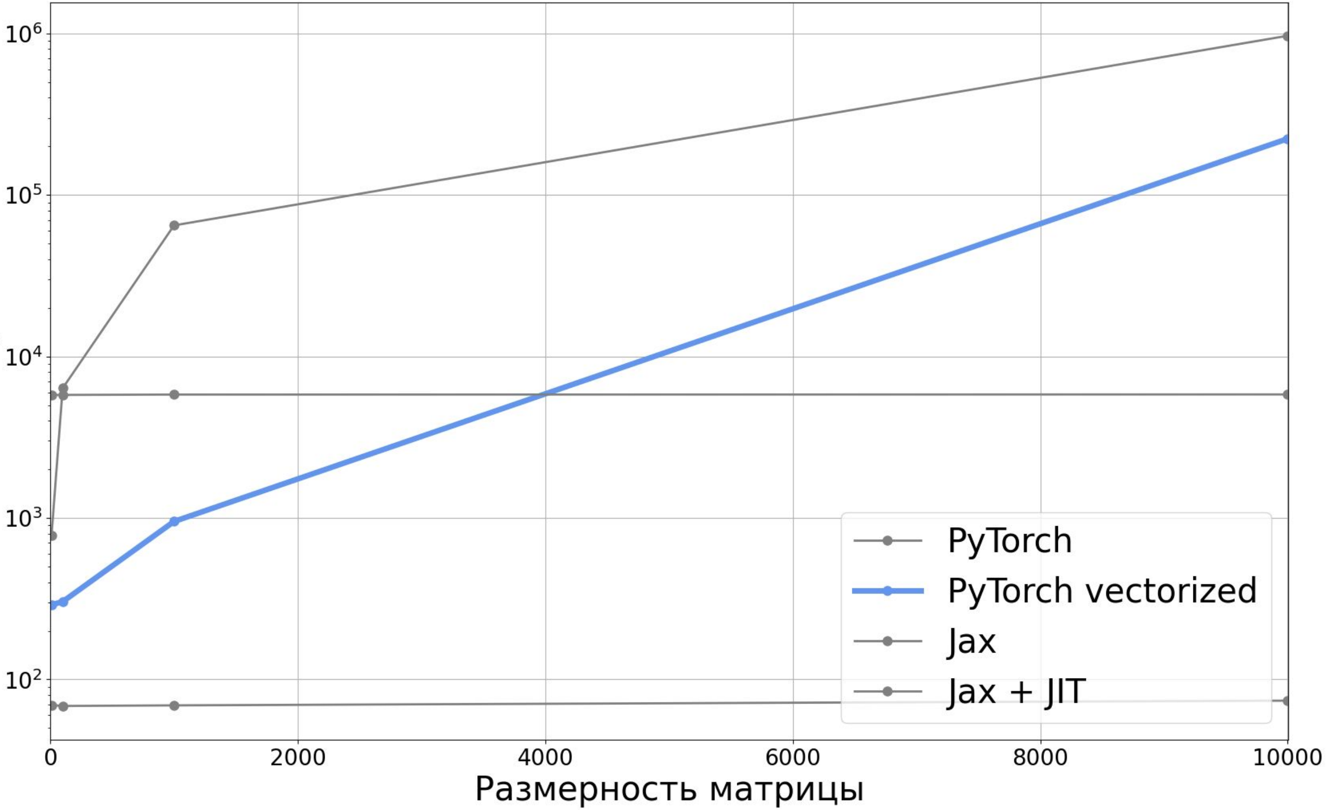
```
def torch_fn(X):
  return torch.sum(torch.mul(X,X))

torch.autograd.functional.hessian(torch_fn, X,
vectorize=False)

def jax_fn(X):
  return jnp.sum(jnp.square(X))

jax_fn = jacfwd(jacrev(jax_fn))
```

Performance benchmarks: PyTorch vs Jax, µs



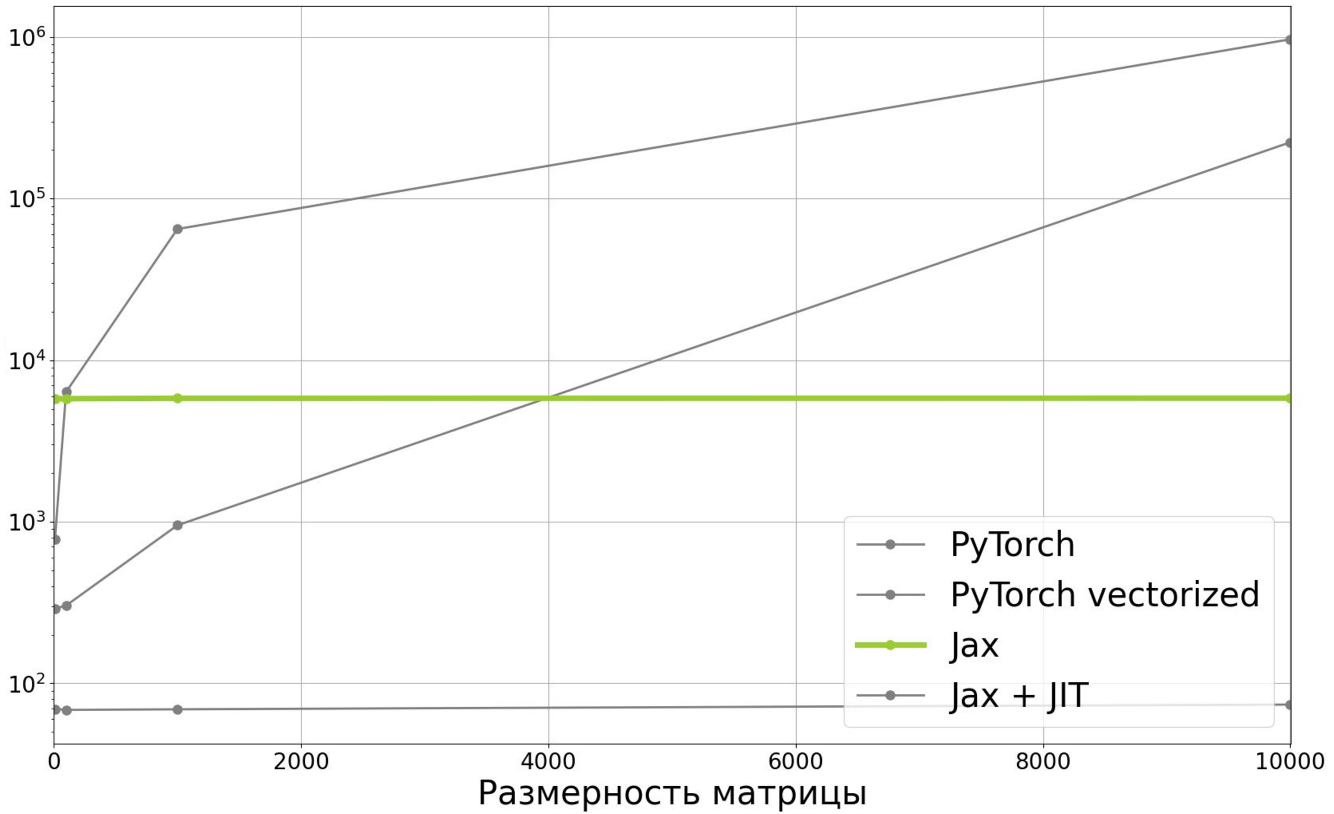
ozon{ech

Performance benchmarks: PyTorch vs Jax, µs

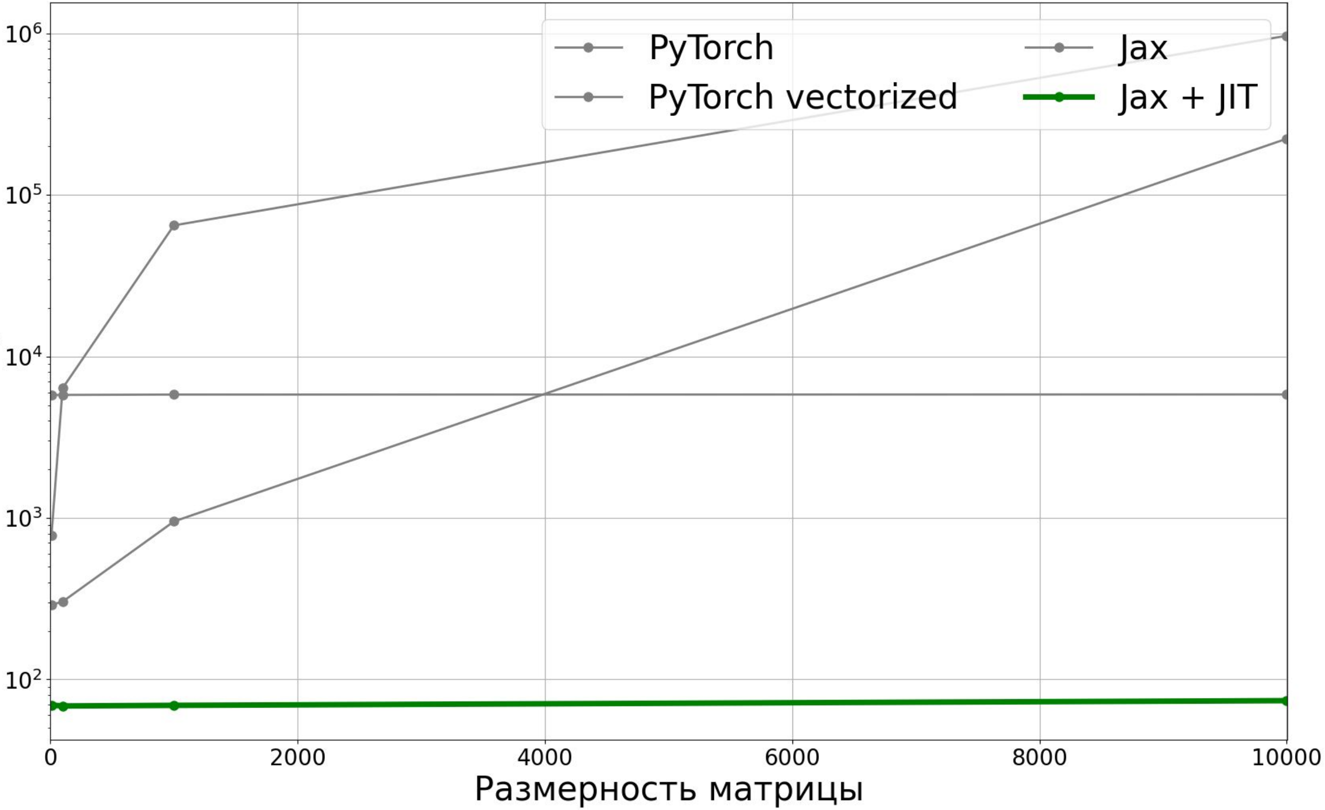


ozon{ech

Performance benchmarks: PyTorch vs Jax, μs

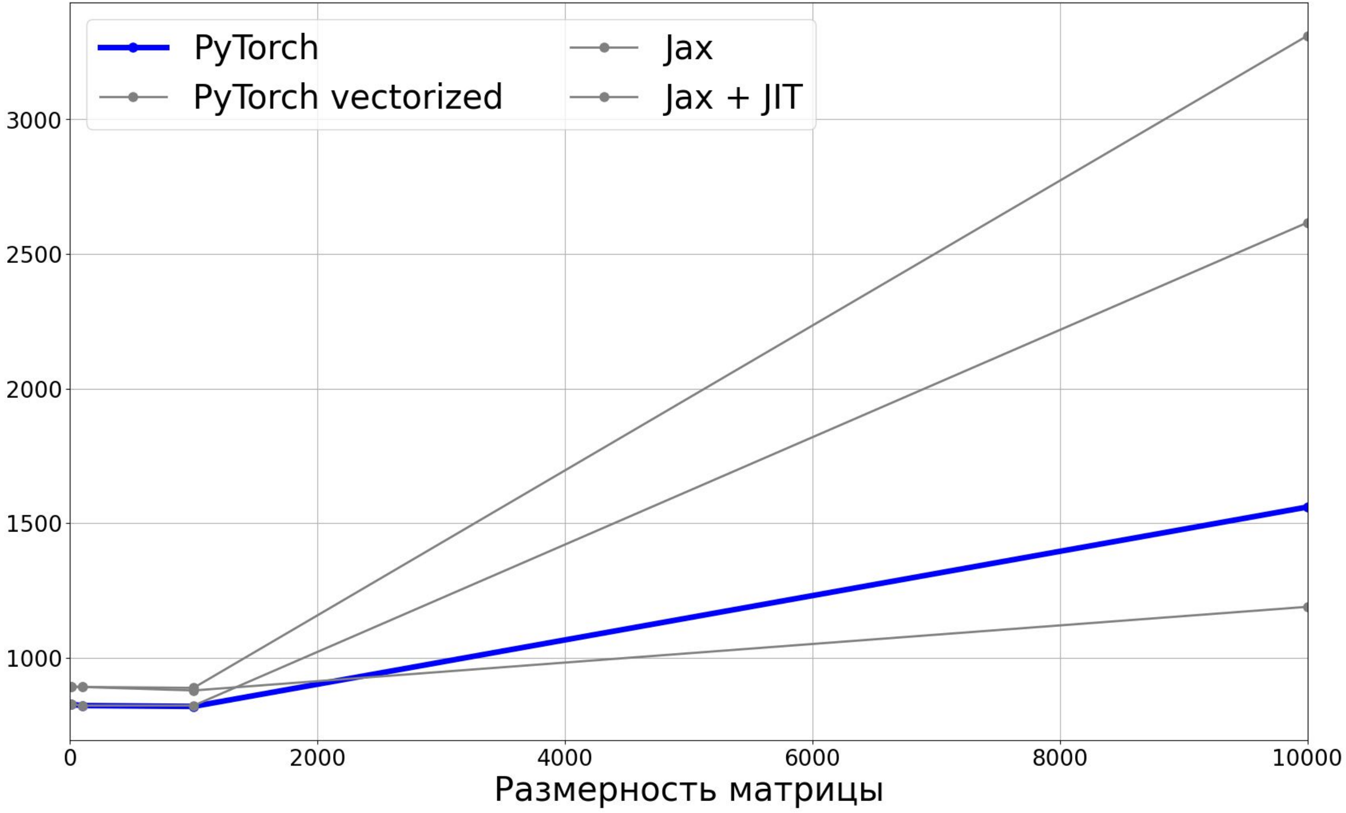


Performance benchmarks: PyTorch vs Jax, µs

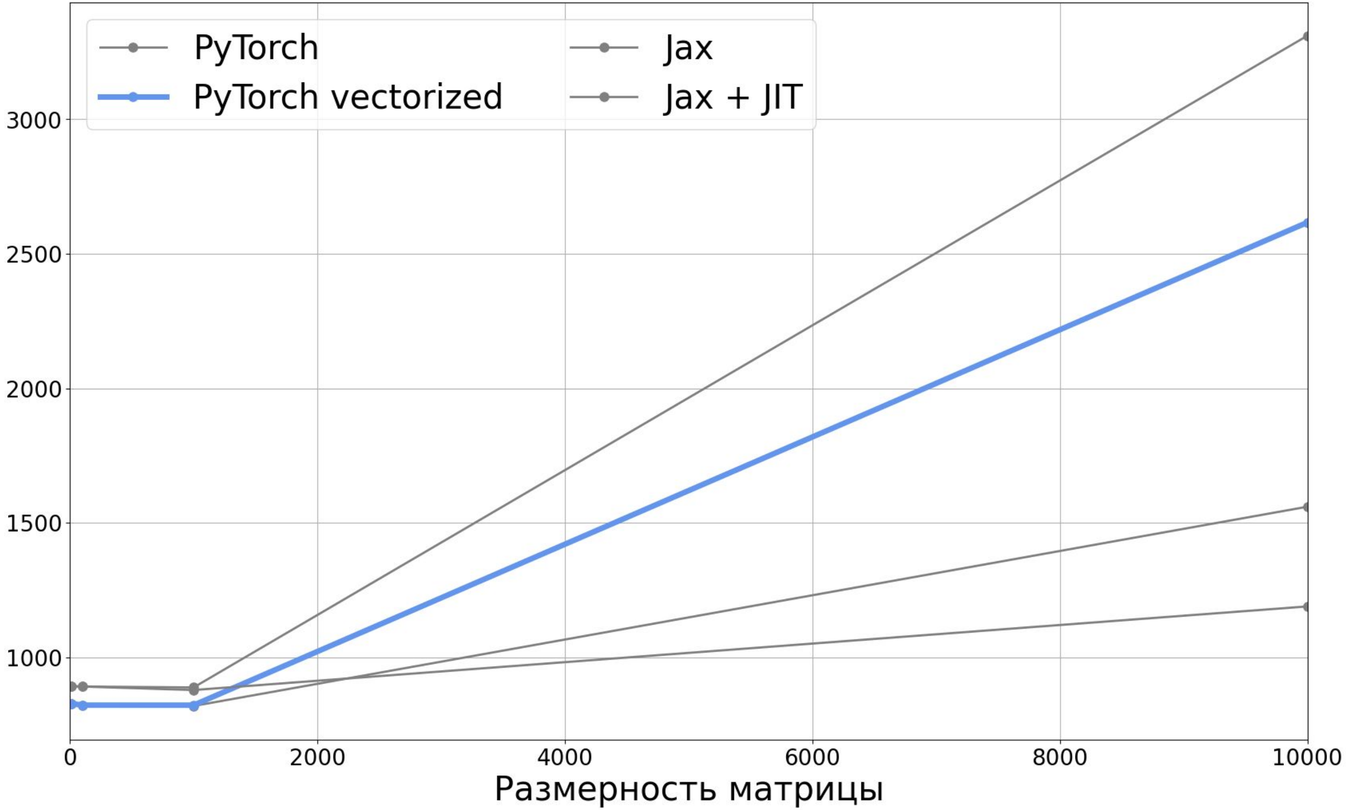


ozon tech

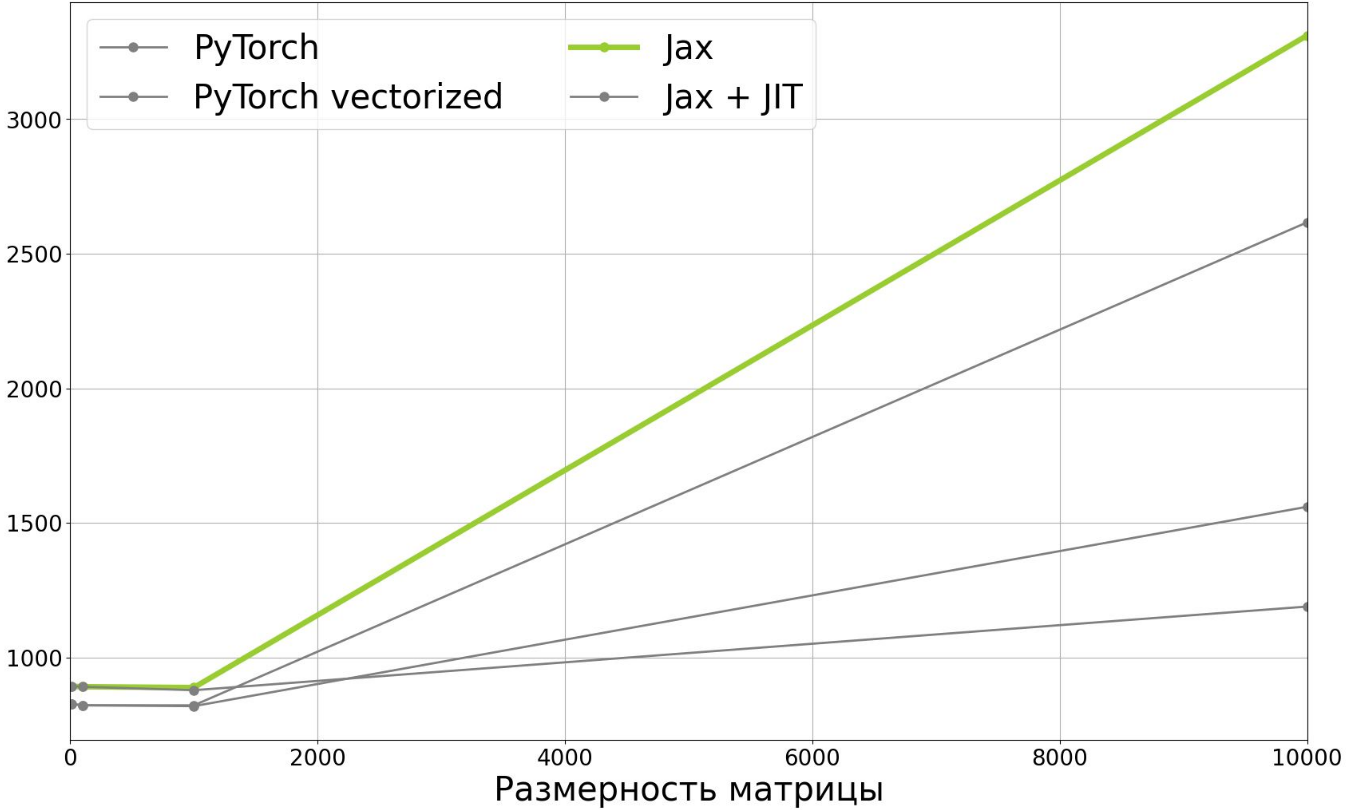
Memory benchmarks: PyTorch vs Jax, RAM MiB



Memory benchmarks: PyTorch vs Jax, RAM MiB

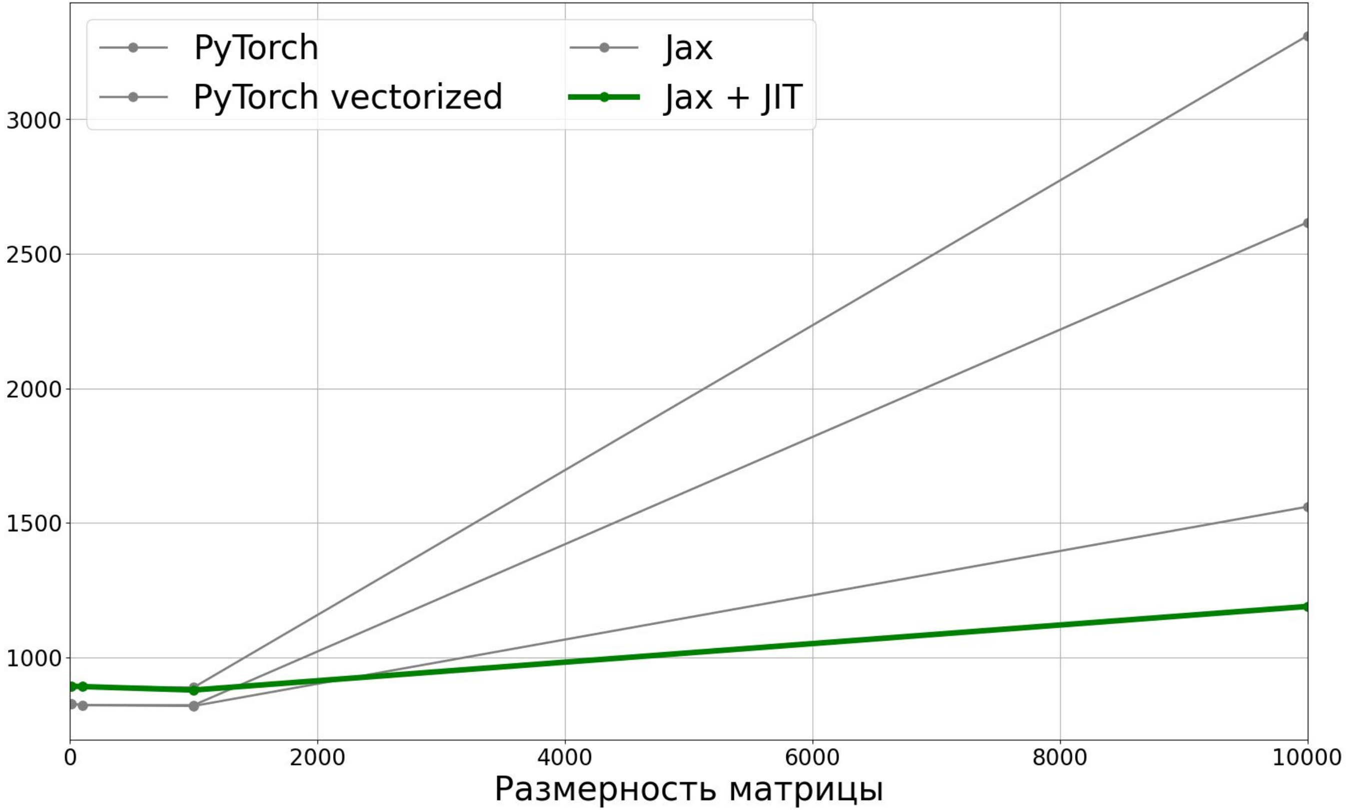


Memory benchmarks: PyTorch vs Jax, RAM MiB



ozon{ech

Memory benchmarks: PyTorch vs Jax, RAM MiB



ozon{ech

Jax vs PyTorch: Обучение нейронных сетей

Генерация датасета

```
np.random.seed(0)
X = np.random.randn(1000, 10)
y = np.random.randint(0, 2, size=(1000,))
…
dataset = torch.utils.data.TensorDataset(X_torch, y_torch)
dataloader = torch.utils.data.DataLoader(dataset,
batch_size=32, shuffle=True)

X_jax = jnp.array(X)
y_jax = jnp.array(y)
```

Обучение нейронных сетей PyTorch

```
class SimpleNN(torch.nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(10, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Обучение нейронных сетей Jax

```
def init_params(key):
    key1, key2 = random.split(key)
    params = {
        'w1': random.normal(key1, (10, 32)),
        'b1': random.normal(key, 32),
        'w2': random.normal(key2, (32, 2)),
        'b2': random.normal(key, 2)
    }
    return params
```

Обучение нейронных сетей Jax

```
…

def forward(params, x):
    x = jnp.dot(x, params['w1']) + params['b1']
    x = jax.nn.relu(x)
    x = jnp.dot(x, params['w2']) + params['b2']
    return x
```

Обучение нейронных сетей Jax

```
@jit
def update(params, x, y, lr=0.01):
    grads = grad(loss_fn)(params, x, y)
    params = {k: v - lr * g for k, v, g in
    zip(params.keys(), params.values(), grads.values())}
    return params
```

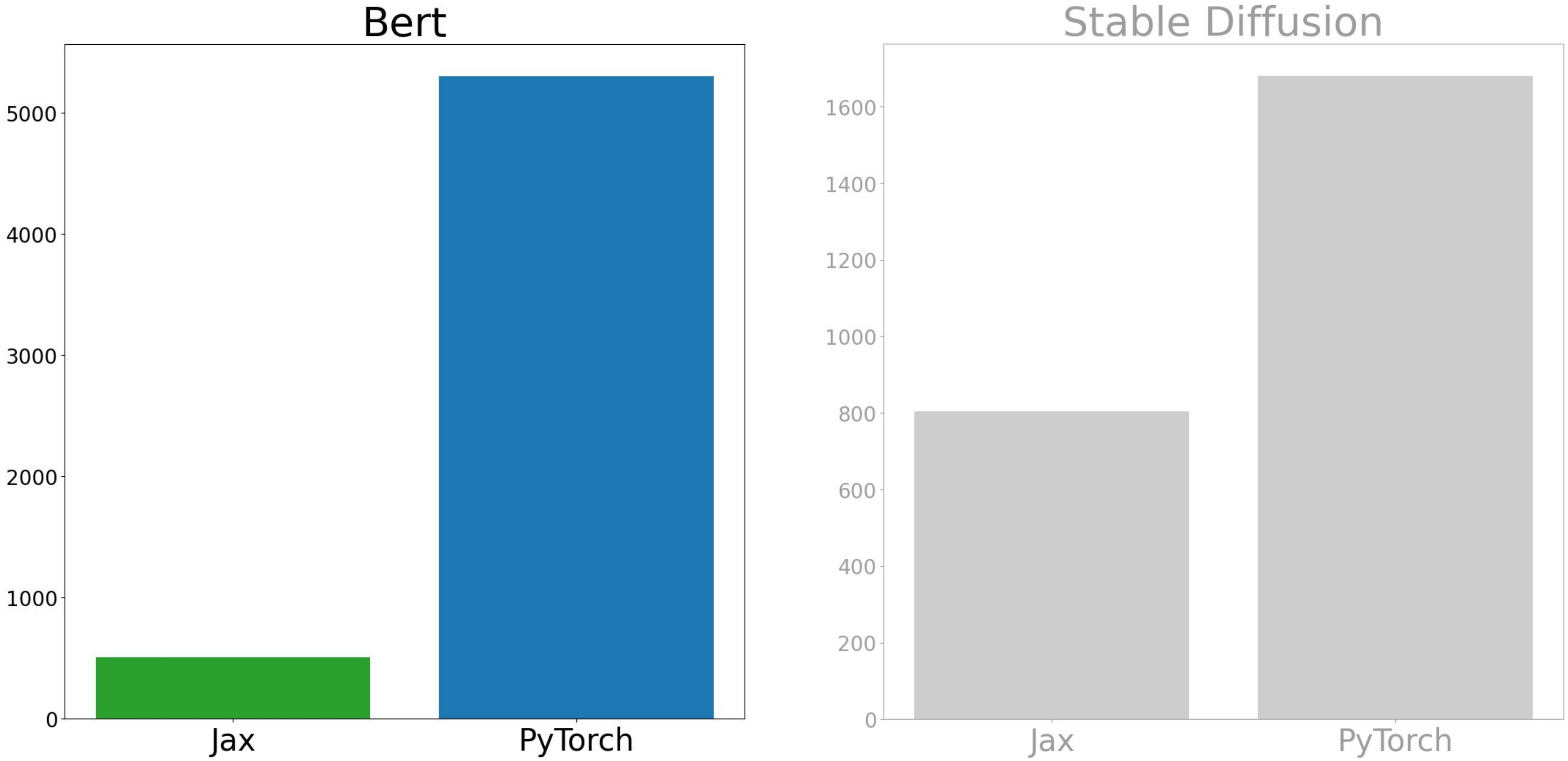
Обучение нейронных сетей Jax + Flax

```python
from flax import linen as nn
import optax

class SimpleNN(nn.Module):
    @nn.compact
    def __call__(self, x):
        x = nn.Dense(32)(x)
        x = nn.relu(x)
        x = nn.Dense(2)(x)
        return x
```

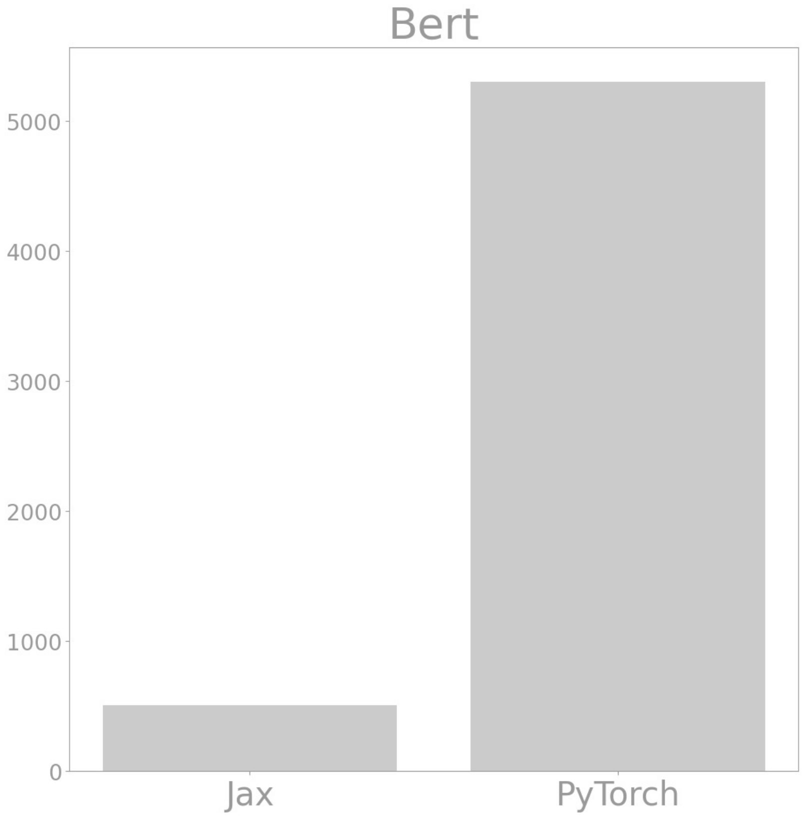
Обучение нейронных сетей Jax + Flax

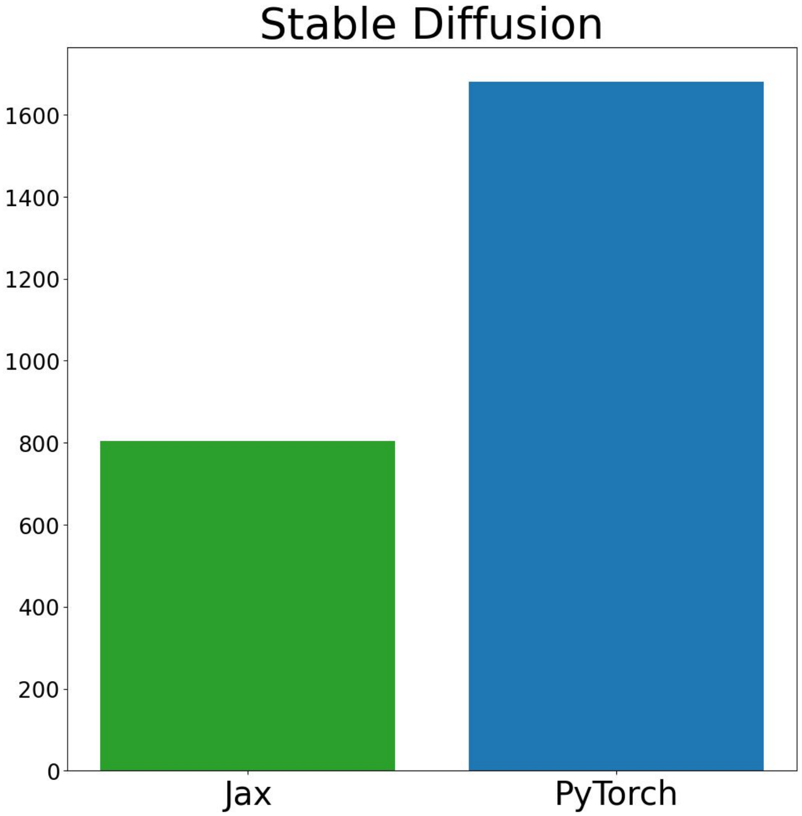
```
@jax.jit
def train_step(params, opt_state, x, y):
    loss, grads = jax.value_and_grad(loss_fn)
                  (params, x, y)
    updates, opt_state = optimizer.update(grads,
                                          opt_state)
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss
```

Model inference: PyTorch vs Jax, μs

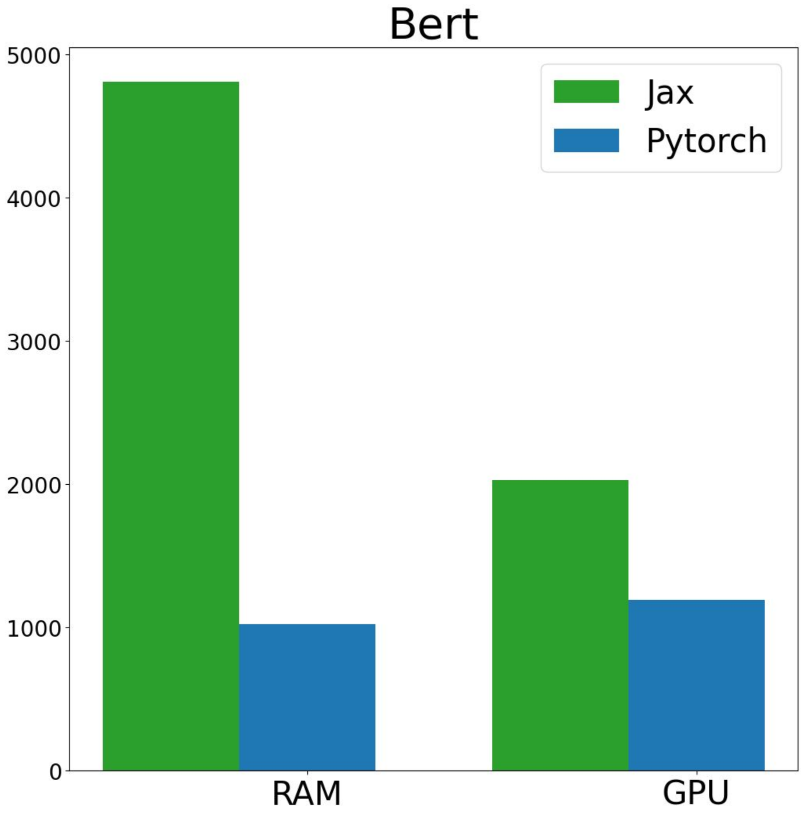


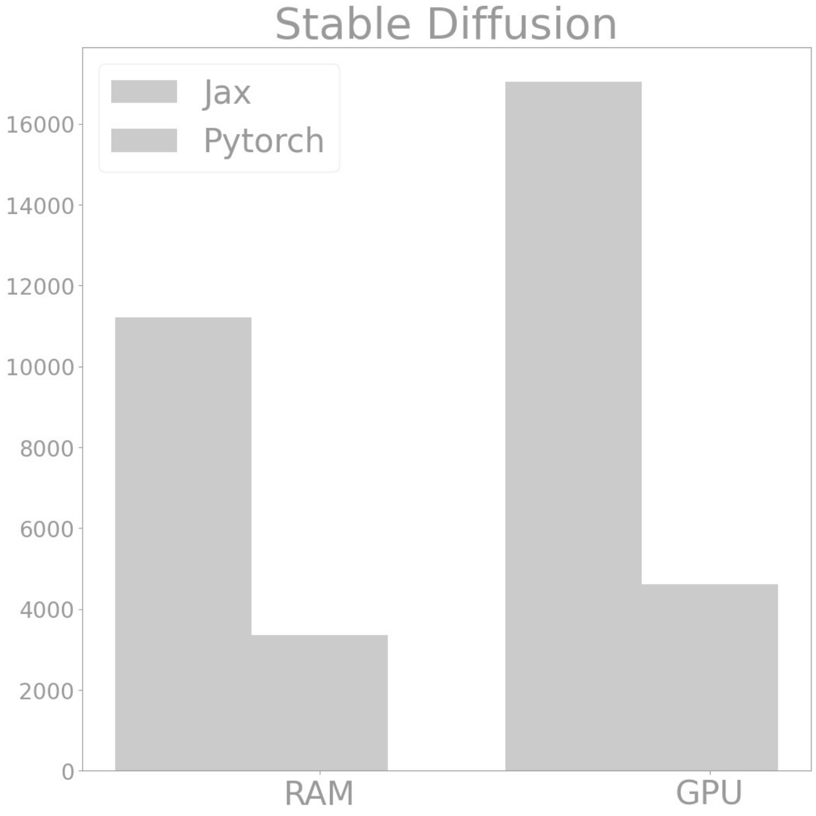
Model inference: PyTorch vs Jax, ms



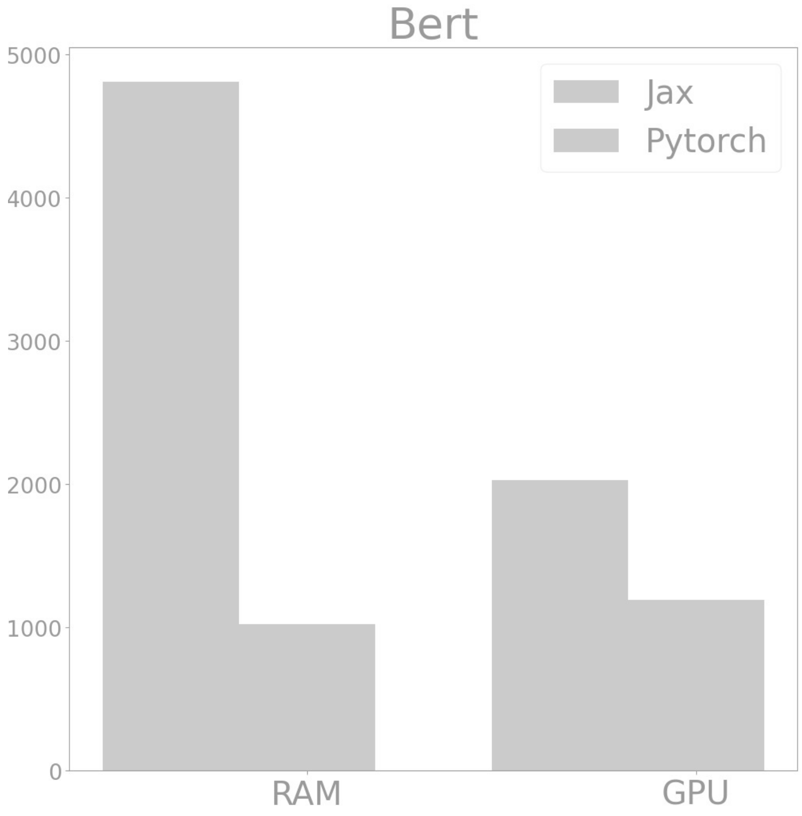


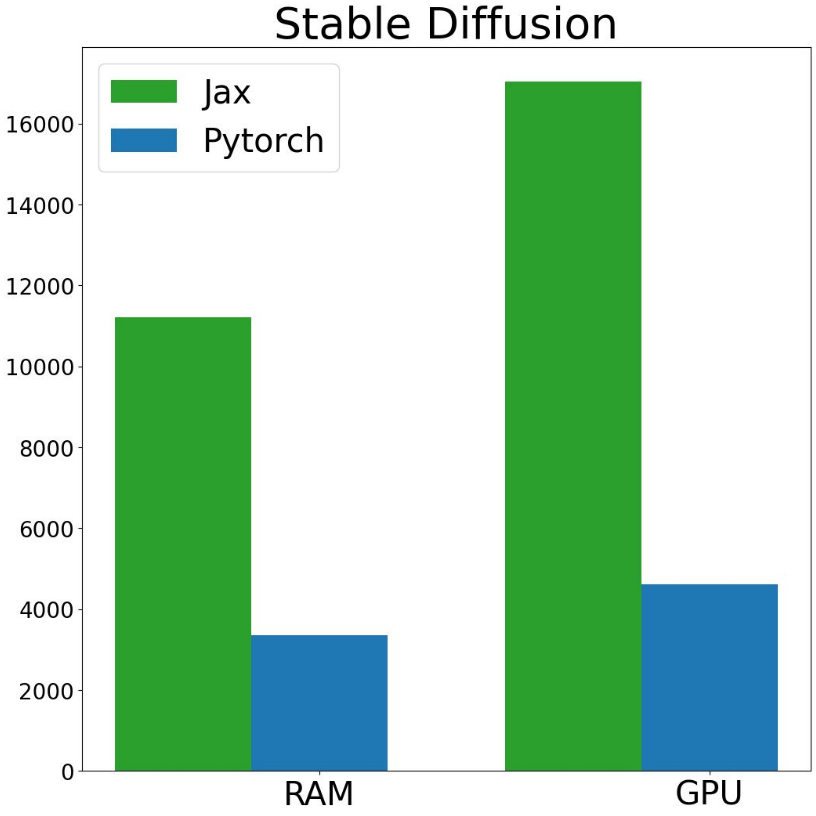
Model inference: PyTorch vs Jax, MiB





Model inference: PyTorch vs Jax, MiB





ozon{ech

03

Экосистема и сообщество Jax

HAIKU

Chex

HAIKU

RLax

Chex

HAIKU

Jraph

RLax

Chex

HAIKU

Jraph

RLax

Chex

Optax

Orbax

Экосистема и сообщество Jax

- Разработка Google Research
- 29.8k stars
- 678 contributors
- https://www.reddit.com/r/JAX/

Спасибо за внимание!

@youngblackwitch

YoungBlackWitch