JAX PROGRAMMING

FROM FUNDAMENTALS TO LARGE-SCALE SYSTEMS

A COMPREHENSIVE GUIDE TO ACCELERATED PYTHON COMPUTING

- HIGH PERFORMANCE COMPUTING
- GPU & TPU ACCELERATION
- DISTRIBUTED & LARGE-SCALE SYSTEMS
- PRACTICAL EXAMPLES & REAL-WORLD CASE STUDIES

RYAN MITCHELL

JAX Programming: From Fundamentals to Large-Scale Systems

A Comprehensive Guide to Accelerated Python Computing

Steve Publications

This book is available at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems

This version was published on 2026-08-02

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

A Comprehensive Guide to Accelerated Python Computing

This book takes you from writing your first JAX program to training large-scale machine learning models on clusters of accelerators. Whether you are a software engineer exploring high-performance computing, a data scientist building modern neural networks, or a researcher pushing the boundaries of scientific simulation, this guide provides the conceptual foundation and practical expertise you need. Through careful explanations, complete working code examples, and real-world case studies, you will learn not only how to use JAX but why it works the way it does and when to reach for each of its powerful tools.

Chapter 1: Why JAX — The Landscape of Python Numerical Computing

The Problem with Existing Tools

In the early 2020s, a researcher at Google wanted to prototype a new neural network architecture. She started by writing NumPy code because it was the most natural way to express her mathematical ideas. The code worked, but it ran on her CPU and took hours for each experiment. She tried moving to PyTorch, which gave her GPU acceleration and automatic differentiation out of the box. But when she wanted to run experiments across multiple GPUs, she found herself wrestling with DataParallel, distributed data loaders, and memory fragmentation issues that slowed her iteration cycle to a crawl.

This scenario repeats itself constantly in research labs and production teams around the world. The core problem is not that existing tools are bad — NumPy, PyTorch, and TensorFlow have all been enormously successful. The problem is that they were built incrementally, each solving immediate needs while inheriting constraints from earlier design decisions, and none of them was designed from the ground up to make a fundamentally different set of tradeoffs.

NumPy is the lingua franca of scientific Python, and for good reason. Its API is clean, its broadcasting semantics are intuitive, and its ecosystem of libraries is unmatched. But NumPy arrays live on the CPU, and while modern CPUs are fast, they cannot compete with GPUs or TPUs for the massive parallel workloads that define modern machine learning and scientific computing. Porting NumPy code to a GPU typically means rewriting it in a completely different framework with its own conventions, APIs, and quirks.

PyTorch solved an important problem by making deep learning feel like writing normal Python code. Its eager execution model lets you use standard Python debugging tools, inspect intermediate values, and write intuitive dynamic computation graphs. But PyTorch's approach to GPU acceleration relies on dispatching operations individually to the device, which introduces

overhead and limits the compiler's ability to optimize across operation boundaries. When you chain together dozens of matrix multiplications, activations, and normalization layers, PyTorch sends each one to the GPU separately, and the GPU spends significant time waiting for the next instruction rather than processing data continuously.

TensorFlow addressed performance through static computation graphs and aggressive optimization, but at the cost of a steep learning curve and painful debugging experience. TensorFlow 2.0 attempted to bridge this gap with eager execution, but the result was a framework that tried to be two things at once – an interactive development environment and a high-performance compilation target – without fully satisfying either goal. The `@tf.function` decorator, meant to compile Python code for performance, became notorious for subtle bugs arising from the interaction between Python semantics and graph compilation.

The fundamental tension here is between flexibility and performance. Interactive, dynamic code is easy to write and debug but hard to optimize. Static, compiled code runs fast but is difficult to express and reason about. Existing frameworks chose different points along this spectrum, but none managed to deliver both genuine ease of use and genuine high-performance compilation for complex workloads.

Enter JAX: A New Paradigm

JAX emerged from Google Brain in 2018 as an experiment in asking a different question. Instead of building another deep learning framework with layers upon layers of abstractions, what if we started with something much simpler: a NumPy-compatible array library that runs on accelerators, combined with a small set of composable program transformations?

The JAX team, including James Bradbury, Roy Frostig, Peter Hawkins, Matthew Johnson, Dougal Maclaurin, and others, built JAX by combining two existing projects. Autograd, created by Dougal Maclaurin and others, provided automatic differentiation for NumPy-like operations. XLA (Accelerated Linear Algebra), Google's internal compiler for linear algebra workloads, provided the hardware acceleration and optimization infrastructure. JAX fused these together with a clean functional programming design philosophy and added two more transformations: just-in-time compilation via `jit` and automatic vectorization via `vmap`.

The result is something that feels fundamentally different from other frameworks. JAX does not try to be a full deep learning framework with built-in optimizers, datasets, or model architectures. Instead, it provides the lowest-level building blocks – arrays and transformations – and lets you compose them however you need. If you want to write a neural network training loop, you write it yourself using `grad` to compute gradients, `jit` to compile the computation, and `vmap` to batch your data. This sounds like more work, but in practice it gives you complete visibility into what is happening and complete control over every aspect of the computation.

The key insight behind JAX's design is composability. Each transformation – jit, grad, vmap, pmap – is a function that takes another function and returns a transformed version of it. These transformations compose naturally. You can write `jax.jit(jax.grad(jax.vmap(loss_fn)))` and get a function that batches over inputs, computes gradients, and runs the whole thing as compiled machine code, all from a single expression. This composability is not an accident; it is the central design principle that makes JAX powerful.

Consider what this means in practice. Suppose you have written a loss function for your model:

```
import jax.numpy as jnp
from jax import jit, grad, vmap

def loss_single(params, x, y):
    """Compute loss for a single training example."""
    prediction = model_forward(params, x)
    return jnp.square(prediction - y)

# Batch over data points
loss_batch = vmap(loss_single, in_axes=(None, 0, 0))

# Compute gradients with respect to parameters
grad_loss_batch = grad(loss_batch, argnums=0)

# Compile the whole thing
train_step = jit(grad_loss_batch)
```

The `train_step` function now takes a batch of data and returns the gradient of the average loss with respect to the model parameters, all computed as optimized machine code on your GPU or TPU. You composed three transformations to get there, each one doing exactly what its name says, and you can reason about each one independently.

What Makes JAX Different

To understand why JAX works the way it does, we need to examine the specific design choices that distinguish it from other frameworks. These choices are not arbitrary; they are deliberate tradeoffs that enable JAX's unique capabilities while introducing constraints you must work within.

Functional purity. JAX transformations require your code to be functionally pure: no side effects, no mutation, no reliance on global state. When JAX compiles a function with `jit`, it does not execute it in the normal Python sense. Instead, it traces the function by running it with special tracer objects that record which operations are performed. Side effects like printing, file I/O, or modifying global variables happen during tracing but not during execution, leading to confusing bugs where your code appears to do nothing when it is actually working correctly.

This constraint feels restrictive at first, especially if you are used to the mutable, imperative style of PyTorch or NumPy. But it is the key enabler of JAX's transformations. If a function has no side effects and always returns the same output for the same input, then JAX can safely compile it, cache it, execute it in parallel, and differentiate through it without worrying about hidden dependencies or ordering requirements. The discipline of functional programming pays dividends in predictability and performance.

Explicit randomness. In NumPy, random number generation relies on a global state that is modified each time you draw a sample. Call `np.random.seed(42)` once, and every subsequent random operation draws from the same sequence. This model breaks down in parallel and compiled code because there is no clear notion of "the next random number" when operations are reordered, fused, or executed concurrently.

JAX takes a completely different approach. Random number generation requires an explicit key that you pass to every random operation. The key is never mutated; instead, you split it to create new independent keys for different purposes. This model makes randomness fully reproducible and composable: you can trace through your code and see exactly which key was used for each random operation, and you can parallelize random operations by splitting a single key into many subkeys.

Immutable arrays. JAX arrays cannot be modified in place. Where NumPy lets you write `arr[0] = 5`, JAX requires you to write `arr.at[0].set(5)`,

which returns a new array with the modification. This immutability is essential for JAX's transformations because it means that operations do not have hidden dependencies on the order in which they are executed. It also enables XLA to optimize memory usage aggressively, reusing buffers and eliminating unnecessary copies.

XLA compilation. Under the hood, JAX uses Google's XLA compiler to turn your Python code into highly optimized machine code for your specific hardware. When you decorate a function with `@jit`, JAX traces it to build an intermediate representation called a jaxpr, which is then compiled by XLA. XLA performs sophisticated optimizations like operator fusion (combining multiple operations into a single kernel to reduce memory traffic), layout optimization (arranging data in memory for efficient access patterns), and parallelization.

The result is that JAX code often runs faster than equivalent code in other frameworks, particularly for large-scale workloads where these optimizations have the most impact. XLA also abstracts away hardware differences: the same JAX code can run on CPUs, NVIDIA GPUs, AMD GPUs, and Google TPUs with minimal changes, because XLA generates appropriate machine code for each target.

Composable transformations. As we discussed earlier, JAX's transformations compose. This is not just a convenient feature; it is the core architectural principle. Each transformation is implemented as a program transformation that manipulates the jaxpr representation of your code. When you stack transformations, they each operate on the intermediate representation produced by the previous one, building up increasingly sophisticated behavior from simple primitives.

This composability enables powerful patterns that would be difficult or impossible in other frameworks. For example, you can compute the gradient of a vectorized function and then JIT-compile the result, all in a single expression. You can parallelize a computation across devices and then differentiate through it to train a model with data parallelism. You can even write custom transformations that build on top of JAX's built-in ones, extending the system to support new kinds of computations.

Who Should Use JAX (and Who Should Not)

JAX is not the right tool for every project or every developer. Understanding its strengths and limitations will help you decide whether it fits your needs.

You should consider JAX if:

- You are doing research in machine learning, particularly novel architectures or training methods that benefit from fine-grained control over the computation graph. JAX's composability and transparency make it ideal for experimentation.
- You are training large models on clusters of GPUs or TPUs. JAX's parallelism primitives and XLA compilation scale efficiently to thousands of devices, and Google uses JAX internally for its largest models including Gemini and PaLM.
- You are doing scientific computing that requires high-performance numerical operations, automatic differentiation, or GPU acceleration. JAX has become a popular choice for computational physics, chemistry, climate modeling, and other domains.
- You value understanding what is happening under the hood. JAX's minimal abstractions mean you always know exactly what your code does, which is invaluable for debugging complex issues and optimizing performance.
- You are comfortable with functional programming concepts and willing to adapt your coding style to embrace immutability, pure functions, and explicit state management.

You might want to look elsewhere if:

- You need a high-level framework with built-in datasets, pretrained models, and one-line training loops. PyTorch with Hugging Face Transformers or TensorFlow/Keras will get you further faster for standard tasks.
- You strongly prefer mutable, imperative code and find functional programming concepts alienating. JAX's constraints will feel frustrating if you are not willing to adapt your mental model.
- You need to deploy models to environments where JAX support is limited. While jax2tf enables export to TensorFlow serving formats, the ecosystem for deploying JAX models directly is less mature than for PyTorch or TensorFlow.
- You are working on a team with limited JAX experience and tight deadlines. The learning curve is steeper than for PyTorch, and the error messages, while improving, can be cryptic when things go wrong.
- You need Windows GPU support. JAX's Windows support is experimental, and GPU acceleration is primarily tested on Linux.

How to Read This Book

This book is structured as a progressive curriculum that builds from fundamentals to advanced topics. Each chapter assumes you have read the previous ones, but you can skip ahead if you are already comfortable with the material. Here is how the chapters are organized:

Chapters 2 through 5 cover the absolute essentials: installation, JAX arrays, JIT compilation, and automatic differentiation. These are the tools you will use in virtually every JAX program, and they form the foundation for everything that follows.

Chapters 6 through 11 dive deeper into JAX's core concepts: functional programming, vectorization with vmap, parallelism with pmap and pjit, control flow, pytrees, and randomness. These chapters explain not just how to use each feature but why it exists and how it fits into JAX's overall design philosophy.

Chapters 12 through 14 focus on machine learning applications: custom gradients, neural networks with Flax and Equinox, and optimization techniques for training at scale. These chapters show you how to build real models and train them efficiently using JAX's ecosystem libraries.

Chapter 15 explores advanced applications beyond standard deep learning, including scientific computing, probabilistic programming, reinforcement learning, and a case study on training large language models.

Chapter 16 covers production concerns: profiling, debugging, memory optimization, deployment, and interoperability with other frameworks.

Throughout the book, code examples are complete and runnable. Each one is carefully explained line by line, with notes about common pitfalls and performance considerations. There are no exercises or quizzes; the goal is to provide a thorough, self-contained reference that you can read cover to cover or consult as needed.

By the end of this book, you will have a deep understanding of JAX's design principles, practical experience with its entire feature set, and the confidence to use it for real projects at research and production scale. Let us get started.

Chapter 2: Getting Started — Installation, Setup, and Your First Program

Python and Environment Prerequisites

Before installing JAX, you need a working Python environment. JAX currently requires Python 3.10 or later, with 3.11 or 3.12 recommended for the best performance and compatibility. You also need NumPy version 1.24 or later, which will be installed automatically as a dependency.

The best practice for managing your JAX environment is to use a virtual environment, which isolates your JAX installation from other Python packages on your system. This prevents version conflicts and makes it easy to reproduce your setup on different machines. The two most common tools for this are venv, which comes bundled with Python, and conda, which provides additional package management features for scientific computing.

To create a virtual environment with venv:

```
python -m venv jax-env
source jax-env/bin/activate    # On Linux/macOS
# or: jax-env\Scripts\activate  # On Windows
```

To create a conda environment:

```
conda create -n jax-env python=3.11
conda activate jax-env
```

Once your environment is activated, you are ready to install JAX. The exact installation command depends on what hardware you want to use.

Installing JAX for Different Hardware

JAX distributes precompiled binary wheels for different hardware configurations. You choose which wheel to install by specifying an extra dependency when you run pip. The core `jax` package contains the Python code, while the companion `jaxlib` package contains the compiled binaries that interface with your hardware. When you specify a hardware-specific extra like `cuda13`, pip installs both jax and the matching version of jaxlib.

CPU-only installation. If you want to get started quickly without worrying about GPU setup, or if you are on a machine without a compatible accelerator, install the CPU version:

```
pip install -U "jax"
```

This installs JAX optimized for your CPU architecture. CPU mode is perfectly fine for learning JAX, running small experiments, and debugging code. Many of the concepts and patterns you learn with CPU-only JAX transfer directly to GPU and TPU usage. The main limitation is speed: operations that take milliseconds on a GPU can take seconds or minutes on a CPU for large arrays.

NVIDIA GPU installation. For NVIDIA GPUs, JAX supports CUDA 12 and CUDA 13. CUDA 13 requires newer hardware (compute capability 7.5 or higher, which includes RTX 30-series and later, as well as data center GPUs like A100 and H100). CUDA 12 supports older GPUs back to compute capability 5.2.

For CUDA 13 on a modern GPU:

```
pip install -U "jax[cuda13]"
```

For CUDA 12 on an older GPU:

```
pip install -U "jax[cuda12]"
```

Before installing, make sure you have the correct NVIDIA driver version. For CUDA 12, you need driver version 525 or later on Linux. For CUDA 13, you need driver version 580 or later. You can check your driver version with `nvidia-smi`. The good news is that JAX bundles the necessary CUDA runtime libraries,

so you do not need to install the full CUDA toolkit separately – the driver is the only prerequisite.

AMD GPU installation. JAX supports AMD GPUs via ROCm on Linux. This requires a compatible AMD GPU and a working ROCm installation. The installation command is:

```
pip install -U "jax[rocm7-local]"
```

AMD support is less mature than NVIDIA support and may have limitations depending on your specific hardware and ROCm version. Consult the JAX documentation for the latest compatibility information.

TPU installation. Google TPUs are primarily available through Google Cloud Platform or the TPU Research Cloud. On a TPU VM, install JAX with:

```
pip install -U "jax[tpu]"
```

TPU support requires running on actual TPU hardware; you cannot run TPU-compiled code on a local machine. If you want to experiment with TPUs without setting up cloud infrastructure, Google Colab offers free TPU runtime that comes with JAX preinstalled.

Mac GPU. As of the current JAX release, there is no native GPU support for macOS. JAX runs on Mac CPUs, including Apple Silicon, but does not accelerate on the integrated GPU. This is a known limitation, and the community has discussed potential paths forward, but for now Mac users should use CPU mode or access GPUs through cloud services.

Nightly builds. If you need the latest features or bug fixes that have not yet made it into a stable release, you can install nightly builds:

```
pip install --upgrade --pre jax jaxlib \
    --find-links
    ↪ https://storage.googleapis.com/jax-releases/jax_nightly_releases.html
    ↪ \
    --find-links https://storage.googleapis.com/jax-releases/jaxlib_nightly_r
    ↪ eleases.html
```

Nightly builds may be unstable and can break between releases, so use them only if you have a specific need.

Verifying Your Installation

After installation, verify that JAX is working correctly by running a simple test. Open a Python interpreter or create a test script:

```
import jax
import jax.numpy as jnp

print(f"JAX version: {jax.__version__}")
print(f"JAX lib version: {jax.lib.jax_version()}")

# List available devices
devices = jax.devices()
print(f"Available devices: {devices}")

# Run a simple computation
x = jnp.array([1.0, 2.0, 3.0])
y = jnp.sin(x)
print(f"sin([1, 2, 3]) = {y}")
```

If you have a GPU or TPU installed correctly, `jax.devices()` should list it alongside the CPU. The output might look like:

```
JAX version: 0.4.35
JAX lib version: 0.4.35
Available devices: [CpuDevice(id=0), CudaDevice(id=0)]
sin([1, 2, 3]) = [0.84147096 0.9092974  0.14112001]
```

If you see only CPU devices when you expected a GPU, check that your NVIDIA driver is installed and up to date, and that the correct CUDA version of JAX was installed. You can verify the CUDA installation by checking the jaxlib version:

```
import jax.lib
print(jax.lib.cuda_version)  # Should match your installation target
```

If you encounter errors during installation, the most common issues are:

- **Driver too old:** Update your NVIDIA driver and retry.
- **Wrong wheel for hardware:** Make sure you installed the CUDA version matching your GPU's compute capability.

- **Conflicting packages:** Remove any existing jax or jaxlib installations before installing a new version. Use `pip uninstall jax jaxlib` first.
- **Python version too old:** Upgrade to Python 3.10 or later.

Your First JAX Array Operations

Now that JAX is installed, let us write some actual code. The most common entry point to JAX is the `jax.numpy` module, which provides a NumPy-compatible API for array operations. If you have used NumPy before, the transition will feel natural. If you have not, the syntax is straightforward and well documented.

Start by importing JAX and jax.numpy:

```
import jax
import jax.numpy as jnp
```

Note that we import `jax.numpy` as `jnp`, not `np`. This convention helps you distinguish JAX arrays from NumPy arrays in your code, which matters because they behave differently in important ways. We will discuss those differences in the next chapter. For now, focus on learning the basic operations.

Creating arrays works just like NumPy:

```
# Create an array from a Python list
a = jnp.array([1, 2, 3, 4, 5])
print(a)        # [1 2 3 4 5]
print(a.dtype)  # int32

# Create a floating-point array
b = jnp.array([1.0, 2.0, 3.0])
print(b.dtype)  # float32

# Create arrays with specific dtypes
c = jnp.array([1, 2, 3], dtype=jnp.float64)
d = jnp.zeros((3, 3))       # 3x3 array of zeros
e = jnp.ones((2, 4))        # 2x4 array of ones
f = jnp.eye(3)              # 3x3 identity matrix
g = jnp.arange(10)          # [0, 1, 2, ..., 9]
h = jnp.linspace(0, 1, 5)   # [0.0, 0.25, 0.5, 0.75, 1.0]
```

Notice that the default floating-point dtype is float32, not float64 as in NumPy. This is intentional: accelerators like GPUs and TPUs are significantly

faster with 32-bit floats, and most machine learning workloads do not need the extra precision of 64-bit floats. If you do need 64-bit precision, you can enable it globally:

```
jax.config.update("jax_enable_x64", True)
```

However, this should be done before any JAX computations, and it comes with a performance cost on accelerators. It is generally better to explicitly specify float64 where needed rather than changing the global default.

Basic arithmetic operations work as expected:

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

print(x + y)   # [5. 7. 9.]
print(x * y)   # [4. 10. 18.]
print(x @ y)   # 32.0 (dot product)

# Matrix multiplication
A = jnp.array([[1, 2], [3, 4]])
B = jnp.array([[5, 6], [7, 8]])
print(A @ B)
# [[19 22]
#  [43 50]]
```

Array manipulation functions are also available:

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

print(arr.shape)        # (2, 3)
print(arr.ndim)         # 2
print(arr.size)         # 6

# Reshaping
reshaped = arr.reshape((3, 2))
print(reshaped)
# [[1 2]
#  [3 4]
#  [5 6]]

# Transposing
transposed = arr.T
print(transposed)
# [[1 4]
```

```
#  [2 5]
#  [3 6]]

# Slicing
print(arr[0, :])   # [1 2 3]
print(arr[:, 1])   # [2 5]

# Indexing with arrays
indices = jnp.array([0, 2])
print(arr[indices])
# [[1 2 3]
#  [4 5 6]] -> actually selects rows 0 and 2, but we only have 2 rows
```

One important difference from NumPy: JAX arrays are immutable. You cannot modify them in place. Where NumPy allows:

```
import numpy as np
arr = np.array([1, 2, 3])
arr[0] = 10  # Modifies arr in place
```

JAX requires a different approach:

```
arr = jnp.array([1, 2, 3])
arr = arr.at[0].set(10)  # Returns a new array; does not modify arr in place
print(arr)  # [10  2  3]
```

The `.at` accessor provides a clean syntax for indexed updates that works consistently with JAX's transformations. We will explore this pattern in more depth later.

Understanding the JAX REPL Experience

A great way to learn JAX is to experiment interactively in the Python REPL or a Jupyter notebook. However, there are some quirks of the JAX execution model that affect how it behaves in interactive environments.

The first thing you will notice is that JAX computations are asynchronous by default. When you call a function that returns a JAX array, you do not immediately get the result. Instead, you get a promise – a DeviceArray object that represents a computation scheduled to run on the device. The actual computation happens later, when the result is needed.

```python
import time
import jax.numpy as jnp

# This returns immediately, even though the computation may take time
start = time.time()
result = jnp.ones((1000, 1000)) @ jnp.ones((1000, 1000))
print(f"Time to schedule: {time.time() - start:.4f} seconds")
# Output: Time to schedule: 0.0003 seconds (essentially instant)

# Force the computation to complete
result.block_until_ready()
print(f"Result shape: {result.shape}")
```

This asynchronous behavior is great for performance – it allows JAX to overlap computation with other work and minimize host-device synchronization overhead. But it can be confusing when benchmarking or debugging. To get accurate timing, always call `block_until_ready()` before measuring elapsed time:

```python
from jax import jit

@jit
def heavy_computation(x):
    for _ in range(100):
        x = jnp.sin(jnp.cos(x))
    return x

x = jnp.ones((1000, 1000))

# Warm up (first call includes compilation time)
heavy_computation(x).block_until_ready()

# Now time the actual execution
start = time.time()
result = heavy_computation(x)
result.block_until_ready()  # Wait for completion
print(f"Execution time: {time.time() - start:.4f} seconds")
```

Another REPL gotcha involves printing. Because JAX arrays are lazy, printing a large array in the REPL triggers its computation. This can slow down your interactive session if you are not careful. For debugging, consider using `jax.debug.print` inside JIT-compiled functions, which we will cover later.

Finally, be aware that JAX preallocates a significant portion of GPU memory when it first runs. By default, it reserves about 90% of available GPU memory.

This is intentional – it avoids fragmentation and allocation overhead during execution. If you need to limit JAX's memory usage, you can configure it:

```
# Limit JAX to use 50% of GPU memory
jax.config.update("jax_platform_name", "gpu")
from jax import config
config.update("jax_gpu_memory_fraction", 0.5)
```

This must be done before any JAX computations, typically at the top of your script.

With JAX installed and verified, and a basic understanding of array operations under your belt, you are ready to dive deeper. The next chapter explores JAX arrays in detail, including all the ways they differ from NumPy arrays and why those differences matter for writing correct and efficient code.

Chapter 3: JAX Arrays — The Foundation of Everything

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Creating and Inspecting Arrays

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Core Array Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Broadcasting and Reshaping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Indexing and Slicing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

JAX vs NumPy Arrays: Key Differences

Memory Layout and Device Placement

Chapter 4: JIT Compilation — Making Python Code Fly

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

What Is JIT Compilation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Using @jit in Practice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Static vs Dynamic Arguments

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Tracing and the XLA Backend

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Common JIT Pitfalls and Solutions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Performance Measurement and Profiling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Chapter 5: Automatic Differentiation — The Engine of Modern ML

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

How Automatic Differentiation Works

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Computing Gradients with jax.grad

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Higher-Order Derivatives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Vector-Jacobian and Jacobian Products

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Value-and-Gradient Computation

Common Autodiff Patterns in ML

Chapter 6: Functional Programming — Embracing JAX Philosophy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Why Functional Programming Matters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Immutability and No Side Effects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Pure Functions and Determinism

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Closures and Function Composition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Thinking Functionally About Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Anti-Patterns to Avoid

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Chapter 7: Vectorization with vmap — Writing Loops Without Loops

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

The Batch Processing Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

How vmap Works

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

In-Axis and Out-Axis Specification

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Nested vmapping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Combining vmap with jit and grad

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

When Not to Use vmap

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Chapter 8: Parallelism — pmap, pjit, and Distributed Computing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Understanding Devices in JAX

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Device Arrays and Placement

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Using pmap for Single-Host Parallelism

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Introduction to pjit

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Shardings and Mesh Definition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Multi-Host Distributed Training

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Chapter 9: Control Flow — Loops and Conditionals That Compile

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Why Python Control Flow Fails Under JIT

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Conditional Execution with lax.cond

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

While Loops with lax.while_loop

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

For Loops with lax.fori_loop

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Scan for Sequential Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Choosing the Right Primitive

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Chapter 10: PyTrees — Structuring Complex Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

What Is a PyTree

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Registering Custom Tree Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Flattening and Unflattening

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Operating on PyTrees

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

PyTrees in Machine Learning Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Advanced PyTree Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Chapter 11: Randomness — Reproducible Stochasticity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Why JAX Handles Randomness Differently

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

The PRNG Key System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Splitting Keys for Parallelism

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

fold_in for Labeling Keys

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Seeding and Reproducibility

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Common Randomness Pitfalls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Chapter 12: Custom Gradients — Beyond Automatic Differentiation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

When You Need Custom Gradients

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Using jax.custom_jvp

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Using jax.custom_vjp

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Gradient Checking Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Practical Examples from Research

Chapter 13: Neural Networks with Flax and Equinox

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Building Blocks of Neural Networks in JAX

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Introduction to Flax

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Introduction to Equinox

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Flax vs Equinox: Choosing a Library

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Common Architectures and Patterns

Chapter 14: Optimization — Training at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Optimizers in JAX Ecosystem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Learning Rate Scheduling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Gradient Clipping and Normalization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Mixed Precision Training

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Checkpointing and Experiment Tracking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Scaling to Large Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Chapter 15: Advanced Topics — Scientific Computing, Probabilistic Programming, and RL

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

JAX for Scientific Computing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Introduction to Probabilistic Programming with NumPyro

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Reinforcement Learning with JAX

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Differentiable Physics Simulations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolargescalesystems.

Case Study: Training a Large Language Model

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolargescalesystems.

Chapter 16: Performance Engineering and Deployment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Profiling JAX Programs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Debugging Techniques and Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Memory Optimization Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Interoperability with NumPy and Other Libraries

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Model Export and Serving

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Production Deployment Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

Conclusion: The Future of JAX and Accelerated Computing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/jaxprogrammingfromfundamentalstolarge-scalesystems.