Data sharding

> ⚠ Warning
>
> Data sharding is an **experimental** feature. It is numerically correct **only** for models built from per-row (*map*) operations and axis-0 reductions (`sum` / `mean` / `logsumexp`). For any model that couples rows across the shard boundary it can crash, deadlock, or — worst — **silently bias** the posterior. Read the Correctness limits section before using it, and audit every run with `m.shard_report()`.

What sharding does

Data sharding is **data parallelism**: BayesForge splits the leading (observation) axis of your data arrays into one slice per device, and XLA computes each device's slice of the log-likelihood concurrently, then aggregates. It is orthogonal to **chain parallelism** (`num_chains` with `chain_method='parallel'`, the default), which puts a whole MCMC chain on each device. Both can be active at once.

Sharding is useful when a single fit is dominated by a large per-observation likelihood and you have several devices to spread it across. It is **not** a general speed-up — models that don't fit the map/reduce pattern below will not shard correctly.

CPU and GPU

The API and model code are identical on both platforms; only the hardware differs.

- **CPU** — JAX exposes as many *virtual* devices as the `cores` you request when constructing `bf`. Slices execute in parallel across XLA partitions / OS threads.
- **GPU** — each physical GPU is one device; slices live in separate GPU memory for true hardware parallelism.

Either way, sharding needs **more than one device**. On a single device it is a no-op and emits a warning.

Enabling sharding

Sharding is a two-step opt-in: build a multi-device mesh at construction, then request it per fit.

```
from BayesForge import bf
import jax.numpy as jnp

# 1. Build a multi-device mesh (CPU: cores>1;  GPU: multiple GPUs)
m = bf(platform='cpu', cores=4)

# ... attach data, define model ...

# 2. Opt in for this fit
m.fit(model, shard=True)
```

- `m.fit(..., shard=True)` — shard eligible arrays for this fit.
- `m.fit(..., shard=False)` — disable (explicit).
- `m.fit(..., shard=None)` — the **default**; falls back to the `BF_SHARD` environment variable. Set `BF_SHARD=1` to make sharding the default for every fit.

There is no construction-time shard flag: the device mesh is built whenever `cores > 1` (CPU) or multiple GPUs are present, but sharding itself is always declared per fit.

What gets sharded automatically

When `shard=True`, each array in the data passed to the model is handled as follows:

- **Sharded** — arrays whose leading dimension is exactly divisible by the device count (so the split needs no padding).
- **Replicated** — scalars, 0-D arrays, and arrays whose leading dimension is *not* divisible by the device count. XLA places a full copy on every device.
- **Left as-is** — arrays you placed yourself with `m.shard()` / `m.replicate()`; an explicit placement is never overridden.

Manual placement

You can control placement explicitly instead of relying on the automatic rules:

```
Y_sharded  = m.shard(jnp.array(Y))       # split along leading axis across devices
K_full     = m.replicate(jnp.array(K))   # identical copy on every device

m.fit(model, obs=dict(Y=Y_sharded), num_chains=m.n_devices, shard=True)
```

- `m.shard(array)` — shard `array` along its leading axis. Returns it unchanged on a single-device setup.
- `m.replicate(array)` — place an identical copy on every device (use for small constants / parameter arrays every device needs in full).

Auditing a run: `shard_report()`

After a sharded fit, inspect exactly what the sharder did:

```
m.fit(model, shard=True)
print(m.shard_report())
```

`shard_report()` returns a dict (or `None` if no sharded fit has run) with the array names in each category — `sharded`, `replicated_not_divisible`, `replicated_by_caller`, `passthrough` — plus `n_devices`. Always confirm the arrays you expect to be sharded actually are, and that nothing landed in `replicated_not_divisible` by accident (e.g. a leading dimension not divisible by the device count).

Correctness limits

Sharding splits axis 0 across devices, so it is only correct when the log-likelihood is a composition of **per-row (map)** operations and **axis-0 reductions** (`sum` / `mean` / `logsumexp`). Any operation that couples rows across the device boundary is **not yet handled** and can crash, deadlock (under vectorized chains), or silently bias the posterior. Known-unsafe patterns:

1. **Cross-position ops:** `diff` / `lag` / `cumsum` / `scan`, AR / Kalman / HMM / RNN / ODE, convolution, sort, FFT.
2. **All-to-all / pairwise:** outer products on axis 0, Gram / distance / kernel matrices, attention, reciprocity (`Y + Y.T`).
3. **Contraction over the sharded axis** (dot products on the wrong side).
4. **Whole-array linear algebra:** `cholesky` / `inv` / `solve` / `qr` / `svd` / `eig` / `det`.
5. **Index / segment ops crossing partitions:** scatter, `segment_sum`, `bincount`, permutation gather.
6. **A leading axis that is not the observation axis** (feature- or time-major arrays).

This rules out, among others, Gaussian processes, phylogenetic / multivariate-normal models, network reciprocity models, and state-space / time-series models. For those, leave sharding off and use **chain parallelism** (`num_chains`) instead. These cases are being addressed incrementally.

> **Note**
>
> On the first sharded fit of a session, BayesForge prints a one-time warning summarizing these limits. It is not an error — it is a reminder to check that your model fits the map/reduce pattern.

Example: end-to-end

```
from BayesForge import bf
import jax.numpy as jnp

# 4 virtual CPU devices
m = bf(platform='cpu', cores=4)

# A large per-observation regression: likelihood is a pure map + axis-0 reduction → shardable
m.data(data_path)

def model(x, y):
    a = m.dist.normal(0, 1)
    b = m.dist.normal(0, 1)
    s = m.dist.exponential(1)
    m.dist.normal(a + b * x, s, obs=y)     # per-row map, summed over axis 0

m.fit(model, shard=True, num_chains=4)
print(m.shard_report())                    # confirm x, y were sharded
m.summary()
```

See Import BF class for the `cores` / `float_precision` construction arguments.