JAXMg Basics

Introduction to sharding and multi-GPU linear solvers

1 Introduction

JAXMg wraps NVIDIA's **cuSolverMg** (multi-GPU solver library) for JAX, enabling distributed linear algebra operations across multiple GPUs. This tutorial covers the essential concepts for using JAXMg effectively.

Function	Solves	Complexity
`potrs`	$Ax = b$ via Cholesky (A symmetric positive definite)	$O(N^3/3)$
`potri`	A^{-1} via Cholesky	$O(N^3)$
`syevd`	Eigendecomposition of symmetric matrix	$O(N^3)$

Table 1: JAXMg distributed linear algebra functions

2 Core Concepts: JAX Sharding

2.1 The Mesh

A **mesh** is a logical n-dimensional grid of devices with named axes. Think of it as organizing your GPUs into a coordinate system that JAX uses to distribute data and computation.

```python
import jax
from jax.sharding import Mesh, NamedSharding, PartitionSpec as P

# 1D Mesh: 4 GPUs in a line, axis named "x"
devices = jax.devices("gpu")  # [gpu:0, gpu:1, gpu:2, gpu:3]
mesh_1d = jax.make_mesh((4,), ("x",))

# 2D Mesh: 4 GPUs as 2×2 grid, axes "S" and "T"
import numpy as np
mesh_2d = Mesh(np.array(devices).reshape(2, 2), ("S", "T"))
```

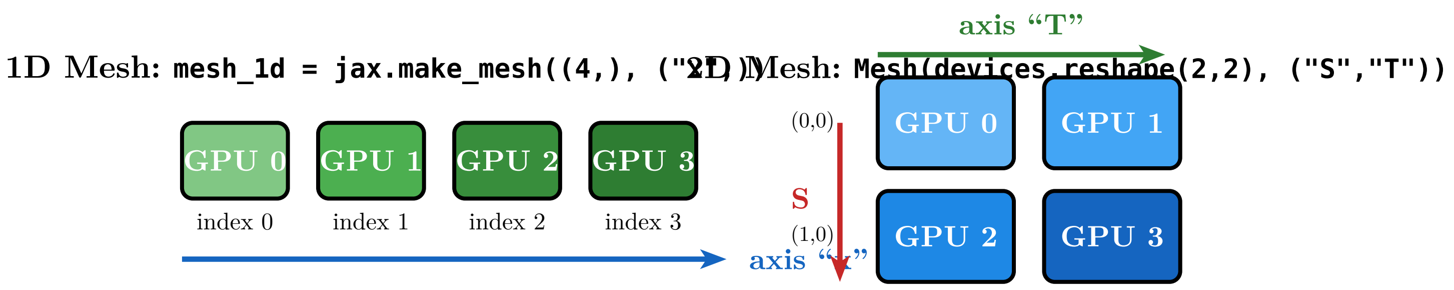

Figure 1: Device mesh configurations. Left: 1D mesh with 4 GPUs along axis "x". Right: 2D mesh with 4 GPUs arranged in a 2×2 grid with axes "S" (rows) and "T" (columns).

2.2 PartitionSpec: How to Shard

`PartitionSpec` (aliased as `P`) tells JAX which mesh axis to shard each array dimension along. `None` means that dimension is replicated (not sharded).

PartitionSpec	Meaning (for 2D array of shape (N, M))
`P("x", None)`	Shard dim 0 (rows) across "x", replicate dim 1 (columns)
`P(None, "x")`	Replicate dim 0 (rows), shard dim 1 (columns) across "x"
`P("S", "T")`	Shard rows across "S", columns across "T"
`P(None, None)`	Fully replicate — each device has a complete copy
`P(("S", "T"), None)`	Flatten S×T mesh into 1D, shard rows across all devices

Table 2: PartitionSpec examples for a 2D array

2.3 Placing Data on Devices

```python
import jax.numpy as jnp

N = 1000
A = jnp.eye(N, dtype=jnp.float64)   # Matrix: 1000 × 1000
b = jnp.ones((N, 1), dtype=jnp.float64)  # Vector: 1000 × 1

# Create mesh and shard data
mesh = jax.make_mesh((4,), ("x",))
A_sharded = jax.device_put(A, NamedSharding(mesh, P("x", None)))  # Row-sharded
b_replicated = jax.device_put(b, NamedSharding(mesh, P(None, None)))  # Replicated
```

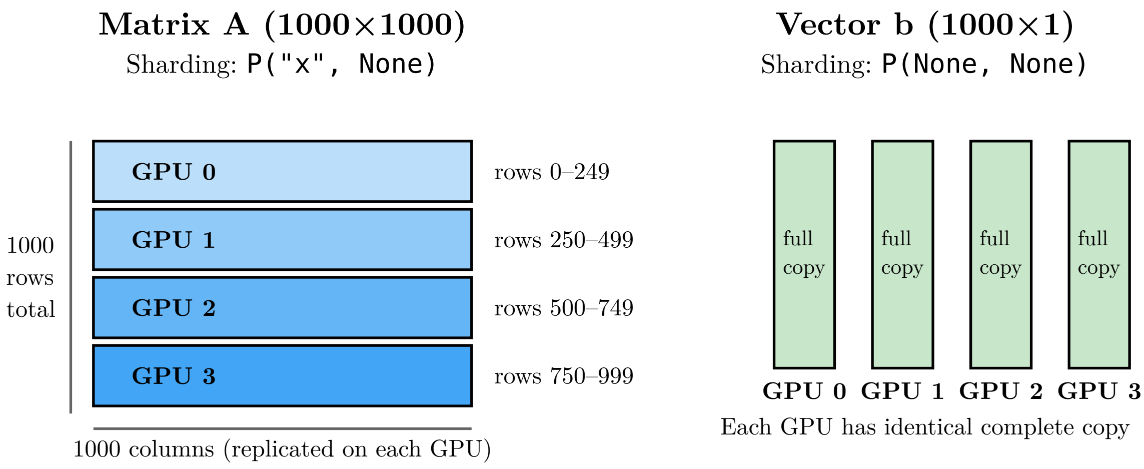

Figure 2: Data distribution across 4 GPUs. Left: Matrix A is row-sharded — each GPU owns 250 rows but all 1000 columns. Right: Vector b is replicated — each GPU has a complete copy.

3 Using `potrs`: Distributed Cholesky Solve

3.1 Basic Usage

```python
from jaxmg import potrs

# Problem: Solve Ax = b where A is symmetric positive definite
```

```python
N = 1000
A = jnp.eye(N, dtype=jnp.float64) * 2  # Diagonal matrix
b = jnp.ones((N, 1), dtype=jnp.float64)

# Setup mesh and sharding
mesh = jax.make_mesh((4,), ("x",))
A = jax.device_put(A, NamedSharding(mesh, P("x", None)))
b = jax.device_put(b, NamedSharding(mesh, P(None, None)))

# Solve!
x = potrs(A, b, T_A=128, mesh=mesh, in_specs=(P("x", None), P(None, None)))
# x ≈ [0.5, 0.5, 0.5, ...]
```

3.2 Key Parameters

Parameter	Description
`T_A`	**Tile size** — block width for distributed algorithm. Larger = faster but more memory. Recommended: 128–2048. Auto-pads if shard size not divisible.
`mesh`	JAX Mesh object defining device topology
`in_specs`	Tuple of PartitionSpecs: `(P("axis", None), P(None, None))` — matrix row-sharded, RHS replicated
`return_status`	If `True`, returns `(solution, status_code)` tuple
`pad`	If `True` (default), auto-pad shards to be divisible by `T_A`

Table 3: `potrs` function parameters

3.3 Blocked Cholesky and Cyclic Column Sharding

cuSolverMg uses a right-looking blocked Cholesky factorization. With tile size `T_A`, the matrix is partitioned into `T_A x T_A` tiles. The `k`-th block column (the panel) consists of tiles `A_(i k)`. Each iteration:

- Factor the diagonal block: $A_{kk} \to L_{kk}L_{kk}^{\dagger}$ (Cholesky on the current tile).
- Panel solve: solve $L_{ik}L_{kk}^{\dagger} = A_{ik}$ for L_{ik} (TRSM on the column panel).
- Trailing update: $A_{ij} = A_{ij} - L_{ik}L_{jk}^{\dagger}$ (GEMM/SYRK on the remaining tiles).

The GPU that owns block column `k` performs the diagonal factorization and panel solve; then all GPUs apply trailing updates to their local tiles.

JAXMg balances this with a 1D block-cyclic column layout: columns are assigned to devices in round-robin tiles of width `T_A` instead of contiguous chunks. This spreads panel ownership, reduces idle time, and avoids a single GPU owning all early columns. Without cyclic sharding, early panels land on one GPU, so others wait for panel data before updating. Internally, JAXMg permutes column tiles into cyclic order using peer-to-peer copies and small staging buffers; the redistribution is deterministic and hidden from the JAX API.

3.4 Internal Workflow

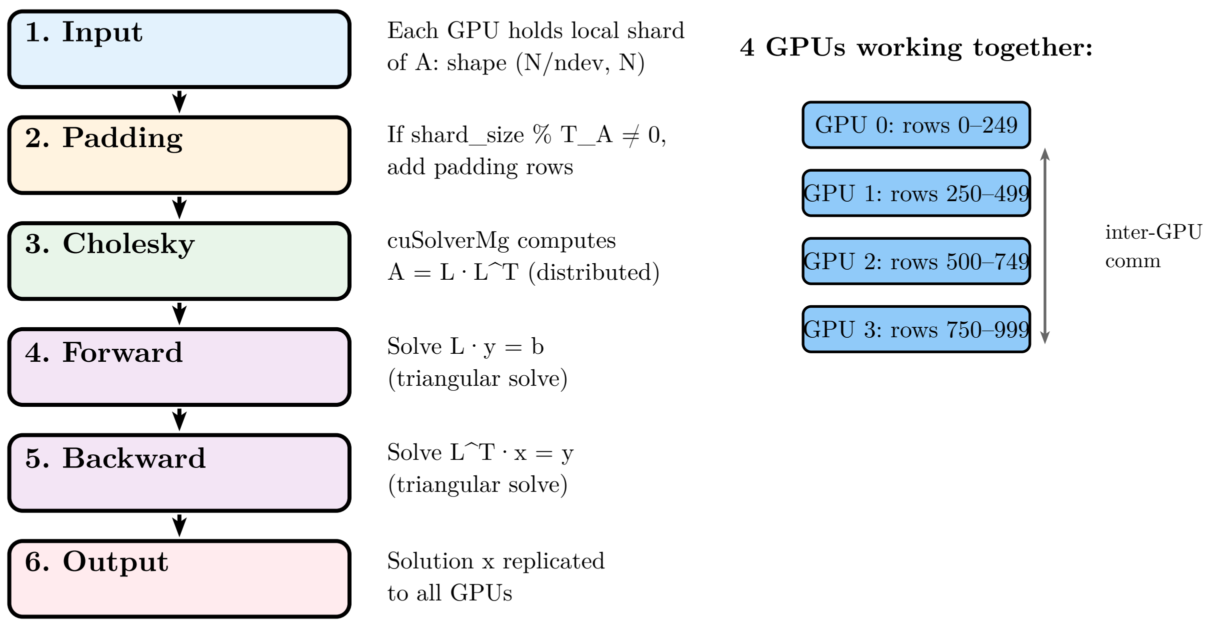
1. Input
Each GPU holds local shard of A: shape (N/ndev, N)
2. Padding
If shard_size % T_A ≠ 0, add padding rows
3. Cholesky
cuSolverMg computes A = L · L^T (distributed)
4. Forward
Solve L · y = b (triangular solve)
5. Backward
Solve L^T · x = y (triangular solve)
6. Output
Solution x replicated to all GPUs
4 GPUs working together:
GPU 0: rows 0–249
GPU 1: rows 250–499
GPU 2: rows 500–749
GPU 3: rows 750–999
inter-GPU comm

Figure 3: Distributed Cholesky solve workflow. The matrix is distributed across GPUs; cuSolverMg handles the distributed factorization and triangular solves with automatic inter-GPU communication.

3.5 Tile Size Selection

The tile size `T_A` controls the block granularity of the distributed algorithm:

```
N = 10000
ndev = 4
shard_size = N // ndev  # 2500 rows per GPU

# Good choices: T_A divides shard_size evenly (no padding needed)
T_A = 250   # 2500 / 250 = 10 tiles ✓
T_A = 500   # 2500 / 500 = 5 tiles  ✓
T_A = 2048  # 2500 / 2048 = 1.22 → padding added automatically ⚠
```

💡 **Tips for choosing `T_A`:**

- Use `T_A ≥ 128` for good performance (small tiles = slow)
- Larger `T_A` → fewer communication rounds, but more memory per tile
- Best: choose `T_A` that divides `N // ndev` evenly to avoid padding overhead
- Common good choices: 256, 512, 1024, 2048

4 The `shard_map` Pattern

4.1 What is `shard_map`?

`jax.shard_map` lets you write code that runs **per-shard** on each device, with explicit collective operations for cross-device communication:

```python
from functools import partial

@partial(jax.shard_map,
         mesh=mesh,
         in_specs=P("x", None),   # Input: row-sharded
         out_specs=P(None))       # Output: replicated
def my_distributed_fn(local_A):
    # This code runs on EACH GPU independently
    # local_A shape: (N/ndev, M) – just this GPU's shard

    my_idx = jax.lax.axis_index("x")  # Which GPU am I? (0, 1, 2, or 3)

    # Collective: sum across all GPUs
    global_sum = jax.lax.psum(local_sum, axis_name="x")

    # Collective: gather all shards
    full_A = jax.lax.all_gather(local_A, axis_name="x", axis=0, tiled=True)

    return result
```

4.2 Common Collective Operations

Operation	Description	Output Shape
`psum(x, "axis")`	Sum values across all devices on axis	Same shape, replicated
`pmean(x, "axis")`	Mean across devices	Same shape, replicated
`all_gather(x, "axis")`	Concatenate shards from all devices	Larger (combined)
`axis_index("axis")`	Get this device's index on axis	Scalar: 0 to ndev-1
`pbroadcast(x, "axis")`	Broadcast from source device	Same on all devices

Table 4: JAX collective operations for `shard_map`

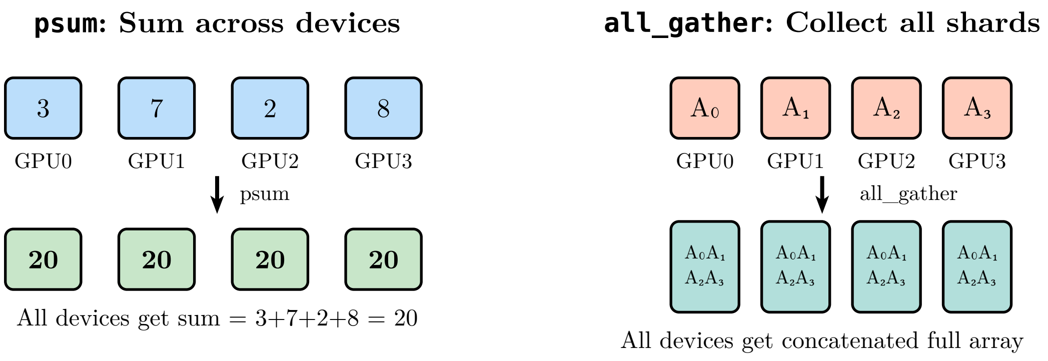

Figure 4: Collective operations illustrated. Left: `psum` sums values and replicates result. Right: `all_gather` collects all shards into complete array on each device.

4.3 Using `potrs_shardmap_ctx`

For advanced use inside your own `shard_map` (when you need custom logic before/after the solve):

```python
from jaxmg import potrs_shardmap_ctx

def custom_solve(local_A, b):
    """Runs inside shard_map context — local_A is just this GPU's shard"""
    local_A = local_A * some_scaling_factor  # Custom pre-processing
    x, status = potrs_shardmap_ctx(local_A, b, T_A=256)  # No extra shard_map
    return x, status

result = jax.shard_map(
    partial(custom_solve, b=b_replicated),
    mesh=mesh,
    in_specs=P("x", None),
    out_specs=(P(None, None), P(None)),
    check_vma=False,  # Required for FFI calls
)(A_sharded)
```

5 Application: MinSR with JAXMg

5.1 The MinSR Algorithm

In Variational Monte Carlo, **minSR** (minimum-step Stochastic Reconfiguration) computes parameter updates efficiently when $N_s \ll N_p$ (fewer samples than parameters):

$$\boxed{\delta\theta = \tau \cdot X^\dagger \big(XX^\dagger + \lambda I\big)^{-1} \cdot E^{\mathrm{loc}}}$$

$X \in \mathbb{R}^{N_s \times N_p}$	Centered Jacobian (samples × parameters)
$T = XX^\dagger \in \mathbb{R}^{N_s \times N_s}$	**Gram matrix** — much smaller than $N_p \times N_p$!
λ	Regularization (diagonal shift, typically 10^{-4} to 10^{-2})
E^{loc}	Centered local energy vector

5.2 Multi-Node Mesh Setup

For clusters with multiple nodes, use a 2D mesh where one axis is within-node and another is across-nodes:

```python
def get_device_grid():
    """Organize devices: rows = local GPUs, columns = nodes"""
    by_proc = {}
    for d in jax.devices():
        by_proc.setdefault(d.process_index, []).append(d)
    hosts = sorted(by_proc)
    return np.array([[by_proc[h][x] for h in hosts]
                      for x in range(jax.local_device_count())]).T

def create_2d_mesh():
    dev_grid = get_device_grid()  # shape: (n_nodes, n_gpus_per_node)
    return Mesh(dev_grid, ("S", "T"))
    # "S" = across nodes, "T" = within node
```

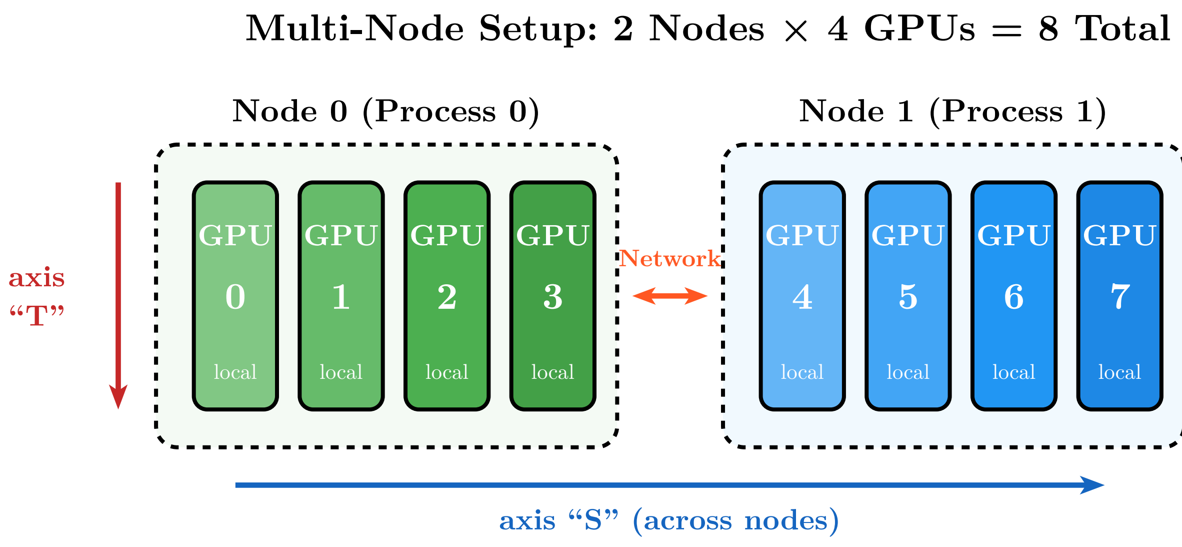

Figure 5: 2D mesh for multi-node cluster. Axis "S" spans across nodes (inter-node communication via network). Axis "T" spans GPUs within each node (fast NVLink/PCIe). Total 8 devices in a (2, 4) grid.

5.3 MinSR Data Flow

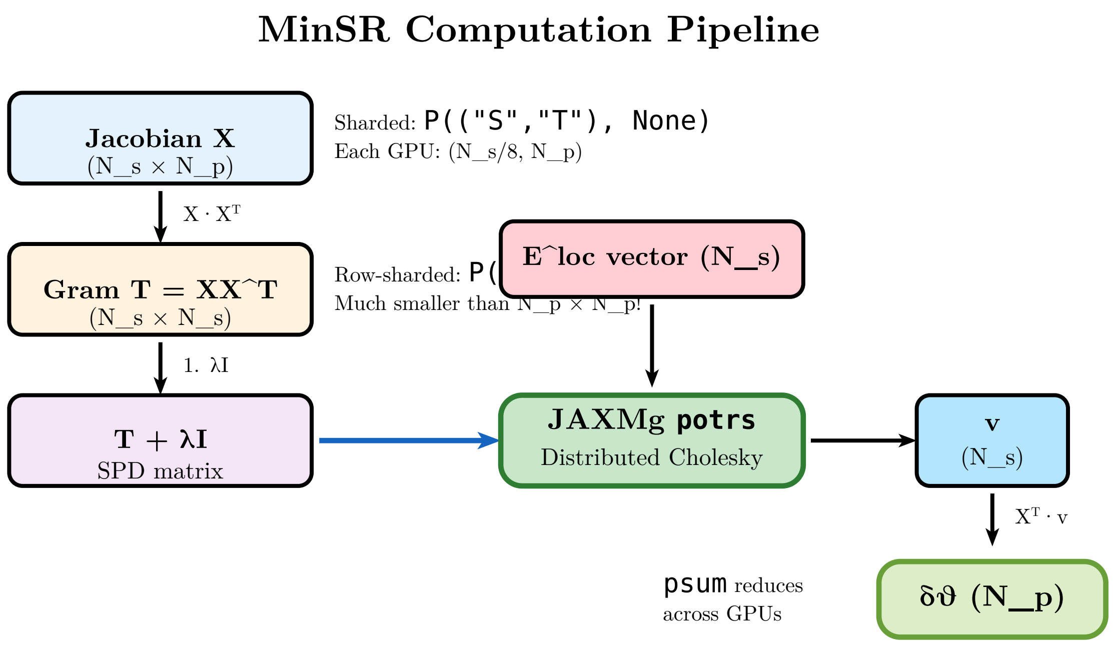

Figure 6: MinSR data flow. The Jacobian X is sharded by samples. The Gram matrix T=XX^T is computed distributedly. JAXMg's `potrs` solves (T+λI)v = E^loc. Final update δϑ = X^T v uses distributed matrix-vector product with `psum` reduction.

5.4 Key Code Snippets

Sharding the Jacobian:

```python
O_LT = jax.lax.with_sharding_constraint(
    O_LT, NamedSharding(mesh_2d, P(("S", "T"), None))
)  # Samples sharded across all 8 GPUs, params replicated
```

Distributed Gram matrix (memory-efficient streaming):

```python
def streamed_gram(O_L, chunk_size):
    """Compute T = X @ X.T without materializing full intermediate"""
    def step(acc, chunk):
        full_chunk = jax.lax.all_gather(chunk, ('S', 'T'), tiled=True)
        return acc + chunk @ full_chunk.T, None
    return jax.lax.scan(step, zeros, O_L_chunked)[0]
```

Distributed Cholesky solve:

```python
v = potrs(T_reg, E_loc[:, None], T_A=2048, mesh=mesh_2d,
          in_specs=(P("T", None), P(None, None)))
```

Final update with reduction:

```python
def update(O_LT, v):
    return jax.lax.psum(v @ O_LT, axis_name=("S", "T"))
delta_theta = jax.shard_map(update, mesh_2d,
    in_specs=(P(("S","T"), None), P(("S","T"))), out_specs=P(None))(O_LT, v)
```

6 Complete Example

```python
import jax
import jax.numpy as jnp
from jax.sharding import Mesh, NamedSharding, PartitionSpec as P
from jaxmg import potrs

# Setup
jax.config.update("jax_enable_x64", True)
devices = jax.devices("gpu")
ndev = len(devices)
mesh = jax.make_mesh((ndev,), ("x",))

# Create Gram matrix (like in minSR: T = X @ X.T)
N_samples, N_params = 4000, 100000
key = jax.random.PRNGKey(0)
X = jax.random.normal(key, (N_samples, N_params)) / jnp.sqrt(N_samples)
T = X @ X.T  # (4000, 4000) — much smaller than (100000, 100000)!

# Add regularization
T_reg = T + 1e-4 * jnp.eye(N_samples)

# RHS (centered local energies)
b = jax.random.normal(key, (N_samples, 1))

# Shard for distributed solve
T_sharded = jax.device_put(T_reg, NamedSharding(mesh, P("x", None)))
b_replicated = jax.device_put(b, NamedSharding(mesh, P(None, None)))

# Distributed Cholesky solve: (T + λI) v = b
v = potrs(T_sharded, b_replicated, T_A=256, mesh=mesh,
          in_specs=(P("x", None), P(None, None)))

# Final minSR update: δθ = X.T @ v
delta_theta = X.T @ v  # (100000, 1)
print(f"Update shape: {delta_theta.shape}")
```

7 Quick Reference

Aspect	Rule
Matrix sharding	Always `P(<axis>, None)` — rows sharded, columns replicated
RHS vector	Always `P(None, None)` — fully replicated on all devices
Tile size `T_A`	Use 128–2048; choose to divide `N // ndev` evenly if possible
Multi-node mesh	2D mesh: `("across_nodes", "within_node")` axes
Memory efficiency	Use streaming Gram computation for large N_params
Output	Solution is always replicated to all devices

Table 5: JAXMg quick reference

✓ Summary

JAXMg makes NVIDIA's multi-GPU linear algebra "just work" with JAX:

1. **Create a mesh** — organize your GPUs into a logical grid
2. **Shard your data** — use `PartitionSpec` to distribute arrays
3. **Call the solver** — `potrs`, `potri`, or `syevd` handle everything

The library manages distributed Cholesky factorization, inter-GPU communication, padding, and memory automatically. You write high-level JAX code; JAXMg handles the CUDA complexity.