

OPENFLUKE · GITHUB.COM/OPENFLUKE/WELVET

Welvet

Feature book — why and what for every engine package, with runnable Go examples.

Version v1.7.0

Generated from welvet/README.md scorecard

Chapters 108

Contents

I · ORIENTATION

1. What Welvet is
2. Repository map

II · FOUNDATION

3. core — types & backends
4. weights — FormatNone MatVec
5. quant — 20 pack formats
6. simd — Plan 9 kernels
7. webgpu — device GEMV & shaders
8. tiling — SC/MC & workgroups
9. architecture — volumetric grid
10. fusedgpu — decoder on device

III · LAYERS

11. layers/dense — MatVec microkernel
12. layers/mha — attention
13. layers/swiglu — gated FFN
14. layers/rmsnorm
15. layers/layernorm
16. layers/embedding
17. layers/softmax
18. layers/sequential
19. layers/residual
20. layers/cnn1 · cnn2 · cnn3
21. layers/rnn · lstm
22. layers/seqmix — mixer contract
23. layers/gdn — gated delta net
24. layers/mamba — selective SSM
25. layers/convt1 · convt2 · convt3
26. layers/kmeans
27. layers/parallel — MoE + cameral
28. layers/metacognition

IV · RUNTIME

29. runtime/forward
30. runtime/backward
31. runtime/training
32. runtime/step — step mesh
65. Cross-numeric train + down-the-dem
67. TrainMode — 29 named updates

V · SYSTEMS

33. systems/dna
34. systems/evolution
35. systems/tween
36. systems/tanhi — TANHI · UDP HUD
37. systems/telemetry
66. lucy — SoftAcc / Score measuring
69. Lucy density — synthetic organism

VI · MODEL IO

38. model/entity — .entity files
39. model/hf — snapshots
40. model/tokenizer
41. model/sampling
42. model/transformer — generate

VII · APPS

43. apps — octo · flux2 · mosstts
44. Octo — model shell
68. Cameral sandwiches + AAI Lucy
70. CamSync — inter-cameral / cross-mesh weight blend

VIII · STUBS

45. stub/seed
46. stub/serialization
47. stub/memory
48. stub/donate
49. stub/fountain
50. stub/hardware
51. stub/accel — NPU/Metal/QNN
52. stub/clustering
53. stub/ensemble
54. stub/evaluation
55. stub/grafting
56. stub/grouping
57. stub/introspection
58. stub/observer
59. stub/pipeline
60. stub/templates
61. stub/universal

IX · VALIDATE

62. w2a — validation harness
63. Validation report — full suite
64. Scorecard → v1.0 / minors

X · NATIVECPU (WELVET.CPP)

76. NativeCPU — opt-in welvet.cpp C ABI
77. NativeCPU — TrainNativeMSE per layer
78. NativeCPU — TrainMode stack / parallel / mesh / line

XI · HEXAGON NPU (WELVET.QNN)

79. QNN — opt-in Hexagon HTP (welvet.qnn)
80. QNN — HTP layer tour (session ABI)
81. QNN — go_v_qnn RESULTS (large shapes)

XII · APPLE ACCELERATION (WELVET.METAL)

82. Apple Metal — mount and forward
83. Apple Metal — Mamba parameter dtypes
84. Apple Metal — training modes
85. Apple Metal — weight return and ENTITY save
86. Apple Metal — JSON backend placement
87. Apple Metal — mixed Go/C++/Metal
88. Apple Metal — Mamba results and stress runner

XIII · NVIDIA ACCELERATION (WELVET.CUDA)

89. NVIDIA CUDA — mount and forward
90. NVIDIA CUDA — Mamba parameter dtypes

91. NVIDIA CUDA — training modes
92. NVIDIA CUDA — weight return and ENTITY save
93. NVIDIA CUDA — JSON backend placement
94. NVIDIA CUDA — mixed Go/C++/CUDA
95. NVIDIA CUDA — results and suite pointers

XIV · AMD ACCELERATION (WELVET.ROCM)

96. AMD ROCm — mount and forward
97. AMD ROCm — Mamba parameter dtypes
98. AMD ROCm — training modes
99. AMD ROCm — weight return and ENTITY save
100. AMD ROCm — JSON backend placement
101. AMD ROCm — mixed Go/C++/ROCm
102. AMD ROCm — results and suite pointers

XV · VULKAN ACCELERATION (WELVET.VULKAN)

103. Vulkan — mount and forward
104. Vulkan — Mamba parameter dtypes
105. Vulkan — training modes
106. Vulkan — weight return and ENTITY save
107. Vulkan — JSON backend placement
108. Vulkan — mixed Go/C++/Vulkan
109. Vulkan — results and suite pointers

XVI · APPLE MLX ACCELERATION (WELVET.MLX)

110. Apple MLX — SyncToMLX and forward
111. Apple MLX — caps and dtype rejects
112. Apple MLX — Dense prepare and Sequential
113. Apple MLX — go_v_mlx results pointers

CHAPTER 1

What Welvet is

github.com/openfluke/welvet ✅ engine

Why it exists

Loom's flat `poly/` package hit import-cycle and honesty walls (QAT morph, silent fallbacks, god-layer). Welvet is the rewrite: one feature per folder, storage-truth dtypes/quants, Dense as the shared MatVec microkernel, tests only in `w2a`, apps only in `apps/`.

What it is

An AI engine in Go: layers, 34 dtypes, 20 quant formats, and Go execution paths (CPU tiled · Plan 9 SIMD · WebGPU), plus optional C++, QNN, Apple Metal, Apple MLX, NVIDIA CUDA, AMD ROCm and Vulkan libraries. Version tracks a 100-point scorecard (today **v1.7.0** · 100/100).

```
Rules
  1. No tests in engine packages          → w2a/
  2. No silent fallbacks                  → hard error
  3. No hardcoded float32                 → Tensor[T]
  4. No QAT                               → DType + Format = truth
  5. Train keeps storage truth            → in-dtype / re-Pack SGD, no retained f32
master
  6. One feature → one folder
  7. v1.0 = scorecard 100/100
```

Non-negotiable engine rules.

v1.0

Engine board is full. w2a `[0] Run ALL` stamped
246,032
matrix cells with
FAIL 0
(RESULT: PASS). GAP cells are declared skips, not silent fails. Apps, stubs, and NPU sit *off* this board — they are sibling trees, not missing Welvet. Training credit (29

> named `TrainMode`s) and cameral sandwiches are first-class; Lucy races live in AAI (chapters 67 · 68 · 70).

Origin

Samuel Watson

Samuel Watson created OpenFluke after intensive T-ALL treatment (post-2018), from a simple frustration: AI tooling was heavy, opaque, and not portable enough to move models cleanly across operating systems or personal devices. Setting up GPU paths often meant large, brittle installs before experimentation could even begin.

That gap drove a long detour through cybersecurity, MBA, and psychology before returning through applied AI study - and finding that most paths still taught framework usage more than system construction. The result was a build journey across planetbridging into OpenFluke projects, from paragon and loom to Welvet.

The early breakthroughs were practical and hard-won: first dense nets in Go, then WebGPU proof points (for example iso-demo v0.1.0), then repeated cycles of speed, layers, 3D, tiling, SIMD, and bit-oriented model paths for personal intelligence.

Why "OpenFluke": open source values plus a life-saving fluke blood test that caught cancer early.
Why "Welvet": a name shaped in gaming sessions with family.

> I wish this tool existed in 2018 to keep trying things in AI and asking "what if I try this?" - and maybe help deterministic medical deployment of AI models.

"Our duty, as soldiers, is to protect humanity. Whatever the cost." - Master Chief

---

Go example

examples/01-welvet/main.go

Run: `cd welvet/examples/01-welvet && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/dense"
```

```
)

func main() {
        l, err := dense.New(8, 4, core.ActivationReLU, core.DTypeFloat32)
        if err != nil {
                panic(err)
        }
        l.Exec.Backend = core.BackendCPUTiled

        x := core.NewTensor[float32](1, 8)
        for i := range x.Data {
                x.Data[i] = float32(i) * 0.1
        }
        pre, post, err := dense.Forward(l, x)
        if err != nil {
                panic(err)
        }
        fmt.Println("pre", len(pre.Data), "post", len(post.Data))
}
```

Output

exit 0 · last run via `go run .`

```
pre 4 post 4
```

Validate (harness)

```
cd welvet && go build ./...
cd w2a && go test ./tests/dense -v
```

CHAPTER 2

Repository map

✅ layout

Why it exists

Readers need a single map of what is engine vs harness vs app vs stub.

What it is

Top-level folders and their ownership. Engine packages never import w2a.

```
welvet/
  core weights quant simd webgpu tiling architecture fusedgpu
  layers/*          ← one folder per op (parallel = MoE + cameral)
  runtime/{forward,backward,training,step}
  systems/{dna,evolution,tween,tanhi,telemetry}
  lucy/             ← Score / Finalize / BuildLPD density (shared by hosts)
  model/{entity,hf,tokenizer,sampling,transformer}
  apps/{octo,flux2,mosstts}
  stub/*            ← designed surfaces, partial or empty
  w2a/              ← harness (separate module)
  tools/            ← eval / test tooling
```

Engine never imports apps. Lucy *measuring* is `lucy/` (public). Lucy *races* (test41 / test48 / test50) live in the AAI tree and `replace` this module — see §68 cameral + AAI. w2a owns every timed matrix, including Test49 (all 23 train modes on origin-only cubes).

Go example

examples/02-tree/main.go

Run: `cd welvet/examples/02-tree && source ../env.sh && go run .`

```
package main

import (
        "fmt"
```

```
        "os"
)

func main() {
        // Mental model only — list engine roots you depend on:
        roots := []string{
                "core", "weights", "quant", "simd", "webgpu", "tiling",
                "architecture", "fusedgpu", "layers", "runtime", "systems", "lucy",
"model",
        }
        for _, r := range roots {
                fmt.Println("import github.com/openfluke/welvet/" + r + "/…")
        }
        _ = os.Getenv // apps/stub/w2a are separate concerns
}
```

Output

exit 0 · last run via `go run .`

```
import github.com/openfluke/welvet/core/…
import github.com/openfluke/welvet/weights/…
import github.com/openfluke/welvet/quant/…
import github.com/openfluke/welvet/simd/…
import github.com/openfluke/welvet/webgpu/…
import github.com/openfluke/welvet/tiling/…
import github.com/openfluke/welvet/architecture/…
import github.com/openfluke/welvet/fusedgpu/…
import github.com/openfluke/welvet/layers/…
import github.com/openfluke/welvet/runtime/…
import github.com/openfluke/welvet/systems/…
import github.com/openfluke/welvet/lucy/…
import github.com/openfluke/welvet/model/…
```

CHAPTER 3

core — types & backends

github.com/openfluke/welvet/core ✅

Why it exists

Every polymorphic path needs one place for DType, LayerType, Activation, Backend, Tensor[T], and slim Layer metadata — without QAT morph defaults.

What it is

34 storage dtypes, Numeric generics, Tensor[T], ExecConfig (BackendCPUTiled | BackendSIMD | BackendWebGPU), Activate/ActivateDeriv, and converters for f16/bf16/fp8/fp4.

Go example

examples/03-core/main.go

Run: `cd welvet/examples/03-core && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/core"
)

func main() {
	t := core.NewTensor[float32](2, 4)
	t.Data[0] = -1
	t.Data[0] = core.Activate(t.Data[0], core.ActivationReLU) // 0

	cfg := core.ExecConfig{
		Backend:   core.BackendSIMD,
		MultiCore: true,
		TileSize:  32,
	}
```

```
        fmt.Println(cfg.Backend.String(), core.ParseDType("bfloat16"), t.Len())
}
```

Output

exit 0 · last run via `go run .`

```
simd bfloat16 8
```

CHAPTER 4

weights — FormatNone MatVec

github.com/openfluke/welvet/weights ✅

Why it exists

Unquantized matrices still need a typed store that streams MatVec and SGD without forcing a float32 master or Morph-as-training.

What it is

Store holds DType + Format + Native/Packed bytes. New[T], MatVec, MatVecT, DecodeRow, SelectWire (F32/F64/I8), ApplySGD. FormatNone: update in native lanes (float32 payload is the only f32 buffer; float64 uses native ALU; other dtypes decode→update→re-encode). Packed: unpack→update→re-Pack then drop scratch. RetainsF32Master() is true only for FormatNone+float32. Dense and composite projs share this store.

Go example

examples/04-weights/main.go

Run: `cd welvet/examples/04-weights && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/quant"
        "github.com/openfluke/welvet/weights"
)

func main() {
        s, err := weights.New[float32](2, 2, []float32{1, 0, 0, 1},
core.DTypeFloat32, quant.FormatNone)
        if err != nil {
                panic(err)
```

```
        }
        y := make([]float32, 2)
        if err := weights.MatVec(s, []float32{3, 4}, y); err != nil {
                panic(err)
        }
        fmt.Println(y) // ≈ [3, 4]
}
```

Output

exit 0 · last run via `go run .`

```
[3 4]
```

CHAPTER 5

quant — 20 pack formats

github.com/openfluke/welvet/quant ✅

Why it exists

Inference, storage, and train need classic Q-packs, k-quants, IQ, Ternary/Binary, and Affine without a separate QAT mode or retained f32 master. Format is storage truth.

What it is

Pack / Unpack / MatVec / MatVecT for FormatNone…AffinePacked. Dense SIMD once-projects codes into Int8QS + scales (EnsureQ* / EnsureK/IQ/AffineSIMDCache) — no full-matrix F32 inflate for k/IQ/Affine. SGD on packed stores: short-lived unpack scratch → update → re-Pack; Packed stays truth.

Families: classic Q8/Q4/Q5 · K-quants · IQ · TernaryPacked/BinaryPacked · AffinePacked.

Honesty: fused Dense SIMD for all 20 quants (group DotKRow / DotIQRow / DotAffineRow + classic Q*/BitNet). Peak dedicated k/IQ Plan 9 `.s` remains scorecard §12.

Go example

examples/05-quant/main.go

Run: `cd welvet/examples/05-quant && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/quant"
)

func main() {
        w := []float32{0.1, -0.2, 0.3, 0.4, -0.5, 0.6, 0.05, -0.15}
        b, err := quant.Pack(quant.FormatQ4_0, w, 2, 4)
```

```
        if err != nil {
                panic(err)
        }
        y := make([]float32, 2)
        if err := quant.MatVec(b, []float32{1, 1, 1, 1}, y); err != nil {
                panic(err)
        }
        fmt.Println("rows", b.Rows, "cols", b.Cols, "y", y)
}
```

Output

exit 0 · last run via `go run .`

```
rows 2 cols 4 y [0.6857143 -1.4901161e-08]
```

CHAPTER 6

simd — Plan 9 kernels

github.com/openfluke/welvet/simd 

Why it exists

CPU peak needs hand-written AVX2/NEON without a silent Go fallback that pretends SIMD ran.

What it is

amd64/arm64 .s kernels: DotTile, DotI8/U8, DotQ4_0, Saxpy, BitNet helpers, packed f16/bf16/fp8/fp4 dots; amd64 AVX2 DotTileF64 (WireF64); Go fused DotKRow / DotIQRow / DotAffineRow for k/IQ/Affine. SimdEnabled() false → BackendSIMD hard-errors.

v1.0.3: Dense FormatNone forward finished the remaining Plan 9 wires — `DotTileF64` (amd64), BF16 convert, expand-once → `DotTile`. Backward still uses saxpy/DecodeRow paths. Universal sizes (any in/out/batch).

dtype family	x86 AVX2	arm64 NEON
float32 DotTile · int8/narrow DotI8 · uint* expand · lowp · nf4/fp6	✓	✓
float64 / int16·32·64 / int / complex* (`DotTileF64`)	✓	× scalar

Arm still runs every FormatNone dtype through the right Dense strategy; the f64 wire is scalar until a NEON `DotTileF64` lands. Hot traffic (f32 / i8) already has NEON.

Go example

examples/06-simd/main.go

Run: `cd welvet/examples/06-simd && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/simd"
)

func main() {
        if !simd.SimdEnabled() {
                fmt.Println("SIMD not available on this arch — BackendSIMD must hard-
error")
                return
        }
        acc := simd.DotTile([]float32{1, 2, 3, 4}, []float32{1, 1, 1, 1}, 0, 4, 0)
        fmt.Println(acc)
}
```

Output

exit 0 · last run via `go run .`

```
10
```

CHAPTER 7

webgpu — device GEMV & shaders

github.com/openfluke/welvet/webgpu ✅

Why it exists

GPU paths must bind a real adapter. Host “fake GPU” was banned so suites cannot stamp WebGPU done when ALU ran on CPU.

What it is

DenseGEMV family (incl. quant/I8/resident), DenseGEMVT/DenseDW, RMSNorm, LayerNorm fwd, Softmax family, SwiGLUFuse. Available()/InitError() gate use.

Go example

examples/07-webgpu/main.go

Run: `cd welvet/examples/07-webgpu && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/webgpu"
)

func main() {
	if !webgpu.Available() {
		fmt.Println("no adapter:", webgpu.InitError())
		return
	}
	y := make([]float32, 2)
	err := webgpu.DenseGEMV(
		[]float32{1, 0, 0, 1},
		[]float32{1.5, 2.5},
		y, 1, 2, 2,
	)
```

```
        fmt.Println(y, err, webgpu.AdapterName())
}
```

Output

exit 0 · last run via `go run .`

```
webgpu probe goos=darwin goarch=arm64 WELVET_WGPU_BACKEND=""
  adapter high-perf: name="Apple M5" vendor="" arch="" driver="" backend=metal
type=integrated-gpu | maxBuf=14302248960 maxSSBO=4294967292 maxSSBO/stage=31
maxBindGroups=8 maxComputeWG=65535 maxInvoc/WG=1024
  → selected high-perf
  adapter low-power: name="Apple M5" vendor="" arch="" driver="" backend=metal
type=integrated-gpu | maxBuf=14302248960 maxSSBO=4294967292 maxSSBO/stage=31
maxBindGroups=8 maxComputeWG=65535 maxInvoc/WG=1024
  adapter force-fallback FAIL: failed to request adapter
  adapter default: name="Apple M5" vendor="" arch="" driver="" backend=metal
type=integrated-gpu | maxBuf=14302248960 maxSSBO=4294967292 maxSSBO/stage=31
maxBindGroups=8 maxComputeWG=65535 maxInvoc/WG=1024
  try RequestDevice inflated: maxBuf=2147483648 maxSSBO=1073741824 maxSSBO/stage=31
maxBindGroups=8 maxComputeWG=65535 maxInvoc/WG=1024
RESULT: OK via=inflated adapter="Apple M5" backend=metal maxSSBO=4294967292
maxBuf=14302248960
[1.5 2.5] <nil> Apple M5
```

CHAPTER 8

tiling — SC/MC & workgroups

github.com/openfluke/welvet/tiling

Why it exists

MatVec throughput depends on tile size and when to go multi-core vs GPU workgroups. Centralizing caps keeps Dense and friends consistent.

What it is

DefaultCPUTile, DefaultGPUWG, CPUTile, PreferMultiCore, GPUWorkgroupsX.

Go example

examples/08-tiling/main.go

Run: `cd welvet/examples/08-tiling && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/tiling"
)

func main() {
        tile := tiling.CPUTile(0) // default 32
        mc := tiling.PreferMultiCore(8, 256, tile)
        wg := tiling.GPUWorkgroupsX(1024, 0)
        fmt.Println(tile, mc, wg)
}
```

Output

exit 0 · last run via `go run .`

```
32 true 16
```

CHAPTER 9

architecture — volumetric grid

github.com/openfluke/welvet/architecture

Why it exists

Networks are spatial (Depth×Rows×Cols×LayersPerCell), not only linear stacks. Topology lives here; compute lives in layer packages.

What it is

Grid/Cell/Coord, NewGrid, BindOp, HopOrder, SetRemoteLink/ResolveHop. VolumetricNetwork is an alias for Grid.

Go example

examples/09-architecture/main.go

Run: `cd welvet/examples/09-architecture && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/architecture"
	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/dense"
)

func main() {
	g := architecture.NewGrid(1, 1, 2, 1) // depth, rows, cols, layersPerCell
	l, _ := dense.New(4, 4, core.ActivationLinear, core.DTypeFloat32)
	if err := dense.Place(g, 0, 0, 0, 0, l); err != nil {
		panic(err)
	}
	_ = g.SetRemoteLink(0, 0, 1, 0, 0, 0, 0, 0) // spatial feedback hop
	fmt.Println("cells", len(g.HopOrder()))
}
```

Output

exit 0 · last run via `go run .`

```
cells 2
```

CHAPTER 10

fusedgpu — decoder on device

github.com/openfluke/welvet/fusedgpu

Why it exists

Token-by-token host round-trips kill decode. A fused engine keeps weights and scratch resident for Q4_0 and BinaryG128 hybrid paths.

What it is

Engine from Spec (AppendTokens/Reset/Close); HybridEngine (PrefillSample/DecodeSample/DecodeChunk). Specs come from model/transformer export.

Go example

examples/10-fusedgpu/main.go

Run: `cd welvet/examples/10-fusedgpu && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/fusedgpu"
	"github.com/openfluke/welvet/model/transformer"
)

func main() {
	m, err := transformer.LoadEntity("model.entity")
	if err != nil {
		fmt.Println("need an ENTITY file:", err)
		return
	}
	spec, err := m.ExportFusedGPUSpec()
	if err != nil {
		panic(err)
	}
	eng, err := fusedgpu.NewFromSpec(spec)
```

```
        if err != nil {
                panic(err)
        }
        defer eng.Close()
        logits, err := eng.AppendTokens([]uint32{1, 2, 3})
        fmt.Println(len(logits), err)
}
```

Output

exit 0 · last run via `go run .`

```
need an ENTITY file: open model.entity: no such file or directory
```

CHAPTER 11

layers/dense — MatVec microkernel

github.com/openfluke/welvet/layers/dense ✅

Why it exists

Most FLOPs are W@x. One Dense stack owns FormatNone×34 and all quants × three backends so every composite proj shares one correctness surface — including native in-dtype SGD.

What it is

New / NewConfigured[T], Forward/Backward (dispatch on Exec.Backend), Place, ApplyGradSGD (→ weights.ApplySGD on the store). SIMD forward (v1.0.3): dtype switch by MatVec strategy — DotTile / DotI8 / lowp packed / expand-once→DotTile / WireF64+DotTileF64; fused Dot* for classic Q*, k/IQ, AffinePacked. Composites (MHA, SwiGLU, CNN im2col, RNN/LSTM/Mamba, residual·sequential·parallel) reuse Dense children via syncProjExec. BackwardSIMD still DecodeRow/saxpy (not the new expand wires).

> **v1.0.3 — FormatNone SIMD forward**
>
> Deep profile (batch=8, 8×256→256, SIMD-only, 34 dtypes): Go vs tiny C++/Rust ports, bit-identical hashes. Go refreshed kernels → suite ~
>
> **4.3×**
>
> geo-mean vs prior Go; wins flipped
>
> **2→24**
>
> (C++ 9, Rust 1). Forward only. Source numbers: welvet-to-rust_n_cpp.

forwardSIMDByWire (strategy groups)

case	dtypes	kernel
narrow i8	int4, int2, ternary, binary	expand → DotI8Tile
uint affine	uint4/2/3/5/6, uint16…uintptr	expand once → DotTile

case	dtypes	kernel
lowp packed	f16, bf16, fp8, fp4	convert tiles → DotTile
expand f32	nf4, fp6, int3/5/6	expand once → DotTile (exact)
stream f64	float64, int16/32/64, int, complex*	WireF64 → DotTileF64
SelectWire	float32, int8, uint8, …	DotTile / DotI8 / affine u8

Before → after (µs/op, Go Δ)

Full Go/C++/Rust columns: before wins **Go 2 / C++ 31 / Rust 1**; after **Go 24 / C++ 9 / Rust 1**. Highlight Go speedups (before→after):

dtype	Go before	Go after	Go Δ	best after
float32	1023.1	**929.5**	1.10×	go
float64	9565.4	**1017.4**	9.40×	go
bfloat16	12538.9	5045.3	2.49×	cpp
int8	708.1	659.3	1.07×	cpp
int64	22271.0	**1059.9**	21×	go
int32	22495.4	**964.1**	23×	go
int16	22147.5	**953.5**	23×	go
int / complex*	~30k	~1.0k	~29–31×	go
uint* / nf4 / fp6 / int3–6	~20–77k	~3.7–10.6k	~5–7×	go
uint8	4322.3	7575.0	0.57×	cpp

C++ still owns int8/narrow, uint8, bf16/fp8. float32 DotTile remains Go's race (**929.5** vs C++ 1162 / Rust 1335). uint8 affine is a known Go follow-up.

Go example

examples/11-dense/main.go

Run: `cd welvet/examples/11-dense && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/dense"
        "github.com/openfluke/welvet/quant"
)

func main() {
        init := make([]float32, 4*8)
        l, err := dense.NewConfigured(8, 4, core.ActivationReLU, core.DTypeFloat32,
quant.FormatNone, init)
        if err != nil {
                panic(err)
        }
        l.Exec.Backend = core.BackendCPUTiled
        x := core.NewTensor[float32](1, 8)
        pre, post, err := dense.Forward(l, x)
        if err != nil {
                panic(err)
        }
        gIn, gW, err := dense.Backward(l, post, x, pre)
        _ = dense.ApplyGradSGD(l, gW, 1e-3)
        fmt.Println(len(gIn.Data), len(gW.Data))
}
```

Output

exit 0 · last run via `go run .`

```
8 32
```

CHAPTER 12

layers/mha — attention

github.com/openfluke/welvet/layers/mha ✅

Why it exists

Transformers need multi-head attention with masks, RoPE/ALiBi, GQA/MQA, and cross-attn — without forking MatVec for every projection.

What it is

Q/K/V/O are Dense children. Presets: DecoderCausal, EncoderBidirectional, CrossAttention, … Attn/RoPE ALU is host today; on-device attn shaders are still open.

Go example

examples/12-mha/main.go

Run: `cd welvet/examples/12-mha && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/mha"
)

func main() {
	l, err := mha.New(mha.DecoderCausal(32, 4, 4))
	if err != nil {
		panic(err)
	}
	x := core.NewTensor[float32](1, 8, 32) // [batch, seq, dim]
	pre, post, err := mha.Forward(l, x)
	fmt.Println(pre != nil, len(post.Data), err)
}
```

Output

exit 0 · last run via `go run .`

```
true 256 <nil>
```

CHAPTER 13

layers/swiglu — gated FFN

github.com/openfluke/welvet/layers/swiglu ✅

Why it exists

Modern decoder FFNs are SiLU(gate)⊙up → down. Projections must share Dense's quant/backend matrix.

What it is

Gate/Up/Down Dense children; DefaultFFN(dModel); WebGPU SiLU⊙ fuse on forward.

Go example

examples/13-swiglu/main.go

Run: `cd welvet/examples/13-swiglu && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/swiglu"
)

func main() {
        l, err := swiglu.New(swiglu.DefaultFFN(64))
        if err != nil {
                panic(err)
        }
        x := core.NewTensor[float32](1, 64)
        _, post, err := swiglu.Forward(l, x)
        fmt.Println(len(post.Data), err)
}
```

Output

exit 0 · last run via `go run .`

```
64 <nil>
```

CHAPTER 14

layers/rmsnorm

github.com/openfluke/welvet/layers/rmsnorm ✅

Why it exists

Llama-style blocks normalize by RMS, not mean+var. Needs native fwd/bwd and WebGPU shaders.

What it is

Per-token RMS + γ on weights.Store; WebGPU fwd+bwd; SIMD DotTile stats + host scale.

Go example

examples/14-rmsnorm/main.go

Run: `cd welvet/examples/14-rmsnorm && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/rmsnorm"
	"github.com/openfluke/welvet/quant"
)

func main() {
	gamma := []float32{1, 1, 1, 1}
	l, err := rmsnorm.NewConfigured(rmsnorm.Config{Dim: 4}, core.DTypeFloat32,
quant.FormatNone, gamma)
	if err != nil {
		panic(err)
	}
	x := core.NewTensor[float32](1, 4)
	_, y, err := rmsnorm.Forward(l, x)
	fmt.Println(y.Data, err)
}
```

Output

exit 0 · last run via `go run .`

```
[0 0 0 0] <nil>
```

CHAPTER 15

layers/layernorm

github.com/openfluke/welvet/layers/layernorm ✅

Why it exists

Classic mean+var normalization with γ/β — still required for many HF architectures.

What it is

WebGPU forward; backward host today. Same dtype×quant axes as other weighted layers.

Go example

examples/15-layernorm/main.go

Run: `cd welvet/examples/15-layernorm && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/layernorm"
)

func main() {
        l, err := layernorm.New(layernorm.Config{Dim: 8})
        if err != nil {
                panic(err)
        }
        x := core.NewTensor[float32](1, 8)
        _, y, err := layernorm.Forward(l, x)
        fmt.Println(len(y.Data), err)
}
```

Output

exit 0 · last run via `go run .`

```
8 <nil>
```

CHAPTER 16

layers/embedding

github.com/openfluke/welvet/layers/embedding

Why it exists

Token IDs must gather rows from a table — not a Dense MatVec — with scatter grads on backward.

What it is

Config{VocabSize, EmbeddingDim, SeqLen}; table on weights.Store; host gather on SIMD/WebGPU today.

Go example

examples/16-embedding/main.go

Run: `cd welvet/examples/16-embedding && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/embedding"
)

func main() {
	l, err := embedding.New(embedding.Config{VocabSize: 32, EmbeddingDim: 16,
SeqLen: 4})
	if err != nil {
		panic(err)
	}
	ids := core.NewTensor[float32](1, 4)
	ids.Data[0], ids.Data[1] = 3, 7
	_, y, err := embedding.Forward(l, ids)
	fmt.Println(y.Shape, err)
}
```

Output

exit 0 · last run via `go run .`

```
[1 4 16] <nil>
```

CHAPTER 17

layers/softmax

github.com/openfluke/welvet/layers/softmax

Why it exists

Classification heads and attention need stable softmax variants, including sparse/Gumbel/Entmax for research paths.

What it is

Weightless layer; KindStandard/Temperature/Grid/Hierarchical/Gumbel/Masked/Sparse/... WebGPU covers std family; exotic kinds hard-error on GPU (no silent host).

Go example

examples/17-softmax/main.go

Run: `cd welvet/examples/17-softmax && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/softmax"
)

func main() {
	l, err := softmax.New(softmax.Config{Dim: 4, Kind: softmax.KindStandard})
	if err != nil {
		panic(err)
	}
	x := core.NewTensor[float32](1, 4)
	copy(x.Data, []float32{2, 1, 0.1, -1})
	_, y, err := softmax.Forward(l, x)
	fmt.Println(y.Data, err)
}
```

Output

exit 0 · last run via `go run .`

```
[0.6380664 0.23473153 0.09543471 0.031767454] <nil>
```

CHAPTER 18

layers/sequential

github.com/openfluke/welvet/layers/sequential ✅

Why it exists

Some cells need an ordered Dense chain without burning grid hops.

What it is

Dense→Dense compose, or mixed Ops via NewFromOps (Dense, SwiGLU, RMSNorm, LayerNorm). Parallel as a child is parallel.ResidualGraft / a Sandwich — Sequential cannot import parallel.

`NewFromOps` walks mixed children (Dense, SwiGLU, RMSNorm, LayerNorm) through the same fwd/bwd as a Dense chain. Parallel is not a Sequential child (import cycle) — wrap F with `parallel.ResidualGraft` or put Parallel in a Sandwich. w2a covers mixed Sequential cells; this is on the v1.0 board, not a leftover "open."

Go example

examples/18-sequential/main.go

Run: `cd welvet/examples/18-sequential && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/sequential"
)

func main() {
        l, err := sequential.New(sequential.Config{Dim: 16, Depth: 3})
        if err != nil {
                panic(err)
        }
        x := core.NewTensor[float32](1, 16)
```

```
        _, y, err := sequential.Forward(l, x)
        fmt.Println(len(y.Data), err)
}
```

Output

exit 0 · last run via `go run .`

```
16 <nil>
```

CHAPTER 19

layers/residual

github.com/openfluke/welvet/layers/residual ✅

Why it exists

Skip connections stabilize deep stacks: y = F(x) + x with correct skip grads.

What it is

F is Dense Dim→Dim, or mixed Ops via NewFromOps (Dense, SwiGLU, RMSNorm, LayerNorm). Parallel as F is parallel.ResidualGraft (y = F(x)+x) — Residual cannot import parallel.

`NewFromOps` builds F over mixed Ops. Parallel as F is `parallel.ResidualGraft` → `y = F(x)+x`. Skip grads still land on F and the identity path. Nested mixed Residual is on the v1.0 board.

Go example

examples/19-residual/main.go

Run: `cd welvet/examples/19-residual && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/residual"
)

func main() {
	l, err := residual.New(residual.Config{Dim: 16, Depth: 2})
	if err != nil {
		panic(err)
	}
	x := core.NewTensor[float32](1, 16)
	_, y, err := residual.Forward(l, x)
```

```
        fmt.Println(len(y.Data), err)
}
```

Output

exit 0 · last run via `go run .`

```
16 <nil>
```

CHAPTER 20

layers/cnn1 · cnn2 · cnn3

github.com/openfluke/welvet/layers/cnn2 ✅

Why it exists

Conv nets must sit on the same dtype×quant×backend matrix as Dense. im2col → Dense GEMV is the intentional first cut.

What it is

cnn1/cnn2/cnn3: host im2col then Proj Dense. Full timed matrices. Tiled conv shaders still .

Go example

examples/20-cnn/main.go

Run: `cd welvet/examples/20-cnn && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/cnn2"
)

func main() {
	l, err := cnn2.New(cnn2.Config{
		InChannels: 1, Filters: 4, Height: 8, Width: 8, Kernel: 3, Stride: 1,
	})
	if err != nil {
		panic(err)
	}
	x := core.NewTensor[float32](1, 1, 8, 8)
	_, y, err := cnn2.Forward(l, x)
	fmt.Println(y.Shape, err)
}
```

Output

exit 0 · last run via `go run .`

```
[1 4 6 6] <nil>
```

CHAPTER 21

layers/rnn · lstm

github.com/openfluke/welvet/layers/lstm ✅

Why it exists

Sequence models before transformers still need vanilla RNN and LSTM with BPTT on the shared MatVec stack.

What it is

IH/HH (and LSTM gates) via Dense; recurrence ALU host; device required for WebGPU path.

Go example

examples/21-rnn-lstm/main.go

Run: `cd welvet/examples/21-rnn-lstm && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/lstm"
        "github.com/openfluke/welvet/layers/rnn"
)

func main() {
        r, _ := rnn.New(rnn.Config{InputSize: 8, HiddenSize: 16, SeqLen: 4})
        l, _ := lstm.New(lstm.Config{InputSize: 8, HiddenSize: 16, SeqLen: 4})
        x := core.NewTensor[float32](1, 4, 8)
        _, yr, _ := rnn.Forward(r, x)
        _, yl, _ := lstm.Forward(l, x)
        fmt.Println(len(yr.Data), len(yl.Data))
}
```

Output

exit 0 · last run via `go run .`

```
64 64
```

CHAPTER 22

layers/seqmix — mixer contract

github.com/openfluke/welvet/layers/seqmix

Why it exists

Attention, SSM, linear attn, and conv mixers must not be accidental forks of mha. Naming the contract keeps packages honest.

What it is

KindAttention | KindSSM | KindLinearAttn | KindConvMix and Contract{Kind,DModel,MaxT}. No compute here.

Go example

examples/22-seqmix/main.go

Run: `cd welvet/examples/22-seqmix && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/layers/seqmix"
)

func main() {
	c := seqmix.Contract{Kind: seqmix.KindAttention, DModel: 64, MaxT: 2048}
	fmt.Println(c.Kind.String()) // attention
}
```

Output

exit 0 · last run via `go run .`

```
attention
```

CHAPTER 23

layers/gdn — gated delta net

github.com/openfluke/welvet/layers/gdn ✅

Why it exists

Linear attention / decode-first mixers (Gated DeltaNet) need a first-class package under KindLinearAttn.

What it is

Exec CPU/SIMD/WebGPU; ForwardDecode; truncated BPTT. Timed matrix is Float32-primary: PermutationOK is f32 × FormatNone/BinaryPacked × CPU/SIMD/WebGPU; other cells GAP (declared), not FAIL.

> **Honesty: GDN GAP ≠ fail**
> `PermutationOK` is Float32 × FormatNone/BinaryPacked × CPU/SIMD/WebGPU only. Exotic dtypes and other packed formats are declared GAP in the timed matrix. Truncated BPTT; blobs are not on the Dense Store dtype axis. Do not read GAP as "GDN is broken."

Go example

examples/23-gdn/main.go

Run: `cd welvet/examples/23-gdn && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/layers/gdn"
)

func main() {
```

```
        l, err := gdn.New(gdn.Config{
                HiddenSize: 64, NumKeyHeads: 4, NumValueHeads: 4,
                KeyHeadDim: 16, ValueHeadDim: 16, ConvKernel: 4,
        })
        if err != nil {
                fmt.Println("config rejected:", err)
                return
        }
        l.Reset()
        x := make([]float32, 64)
        y := make([]float32, 64)
        fmt.Println(l.ForwardDecode(x, y))
}
```

Output

exit 0 · last run via `go run .`

```
<nil>
```

CHAPTER 24

layers/mamba — selective SSM

github.com/openfluke/welvet/layers/mamba

Why it exists

SSM mixers (KindSSM) are not MHA clones — they need their own selective-scan path.

What it is

InProj → softplus(Δ) scan → OutProj. Full timed matrix + train grids (scan ALU host).

Go example

examples/24-mamba/main.go

Run: `cd welvet/examples/24-mamba && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/mamba"
)

func main() {
	l, err := mamba.New(mamba.Config{DModel: 32, DState: 16, Expand: 2, SeqLen:
8})
	if err != nil {
		fmt.Println(err)
		return
	}
	x := core.NewTensor[float32](1, 8, 32)
	_, y, err := mamba.Forward(l, x)
	fmt.Println(y != nil, err)
}
```

Output

exit 0 · last run via `go run .`

```
true <nil>
```

CHAPTER 25

layers/convt1 · convt2 · convt3

github.com/openfluke/welvet/layers/convt1 ✅

Why it exists

Generators and U-Nets need transposed convolution as a peer of CNN, on the same Dense proj surface.

What it is

Scatter upsample + Dense Proj. Full timed matrix + train grids (tiled transpose shaders still §12).

Go example

examples/25-convt/main.go

Run: `cd welvet/examples/25-convt && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/convt1"
)

func main() {
        l, err := convt1.New(convt1.Config{InChannels: 4, Filters: 2, SeqLen: 8,
Kernel: 3, Stride: 2})
        if err != nil {
                panic(err)
        }
        x := core.NewTensor[float32](1, 4, 8)
        _, y, err := convt1.Forward(l, x)
        fmt.Println(y.Shape, err)
}
```

Output

exit 0 · last run via `go run .`

```
[1 2 17] <nil>
```

CHAPTER 26

layers/kmeans

github.com/openfluke/welvet/layers/kmeans ✅

Why it exists

Soft clustering as a differentiable layer lets topology experiments sit inside the same train loop.

What it is

Centers on Dense (K×FeatureDim); soft assignment outputs. Full timed matrix + train grids.

Go example

examples/26-kmeans/main.go

Run: `cd welvet/examples/26-kmeans && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/kmeans"
)

func main() {
	l, err := kmeans.New(kmeans.Config{NumClusters: 4, FeatureDim: 8})
	if err != nil {
		panic(err)
	}
	x := core.NewTensor[float32](1, 8)
	_, y, err := kmeans.Forward(l, x)
	fmt.Println(y.Shape, err)
}
```

Output

exit 0 · last run via `go run .`

```
[1 4] <nil>
```

CHAPTER 27

layers/parallel — MoE + cameral

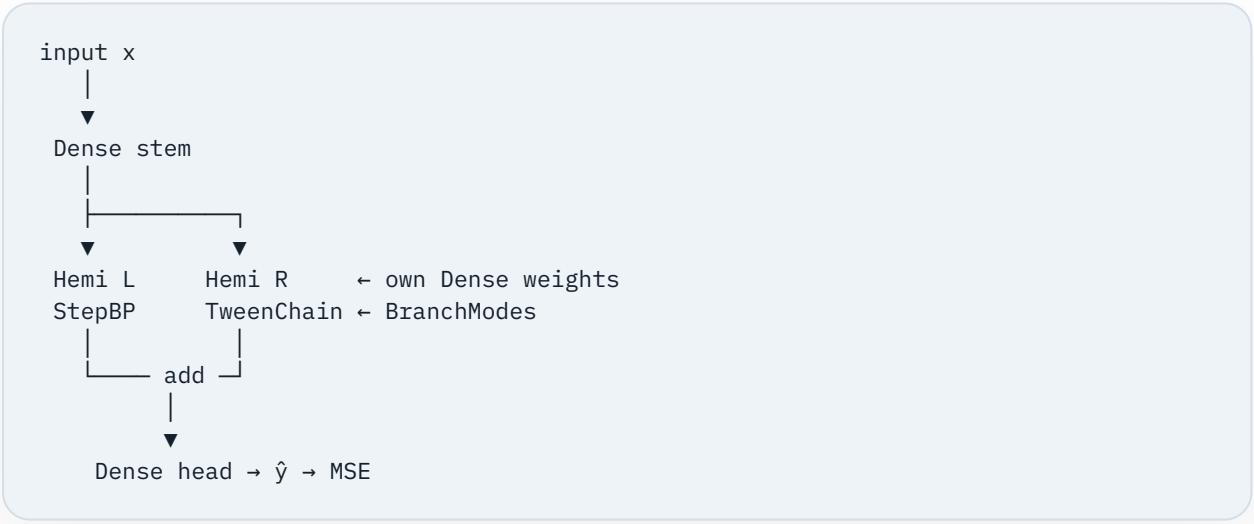
github.com/openfluke/welvet/layers/parallel

Why it exists

Mixture-of-experts and multi-path cells need concat/add/avg/filter combines. Cameral graphs need sibling hemispheres that share input, merge outputs, and optionally train under distinct modes.

What it is

Parallel branches + Stack sandwiches. Hemispheres / Bicameral / Sandwich build nested multi-cameral nets. SetBranchModes + TrainStackMSE let each hemi use a different TrainMode (all 29 named updates — BP, Tween, Split, FastProxy, Sparse, Step*, Mesh*, ...).



Cameral Mix: one TrainMode per hemisphere; one loss on the merge.

Cameral API (v1.0)

- `Hemispheres` / `HemispheresFrom` — n twins (Dense or mixed Ops), merged by add/avg/concat/filter
- `Bicameral` / `Sandwich` / `BicameralFrom` — stem → Parallel → head Stack
- `SetBranchModes(...)` — stamp per-hemi `TrainMode`
- `TrainStackMSE` — forward → MSE → per-branch update (honours BranchModes)

- `ResidualGraft` — skip around a Parallel F without Residual importing this package
- `WriteCameralFile` / `LoadCameral` — cameral `.entity` (in `model/entity`)
- `CamSyncConfig` / `SetCamSync` / `SyncNow` — soft/hard inter-cameral weight blend + cross-layer same-shape pairs → §70

Uniform Bi/Tri/Quad can train via Grid `training.Step` (one mode for the net). Mix jobs need the Stack path so each cameral actually uses its own mode.

Why hemispheres exist, and how AAI Lucy benches use them: §68. Inter-cameral weight sync (crazy cross-mesh same-shape blends): §70. All 29 named updates and their equations: §67.

Go example

examples/27-parallel/main.go

Run: `cd welvet/examples/27-parallel && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/parallel"
        "github.com/openfluke/welvet/quant"
)

func main() {
        // Dense stem → 2 hemispheres (add) → Dense head
        s, err := parallel.Bicameral(8, 16, 1, core.ActivationLeakyReLU,
                core.DTypeFloat32, quant.FormatNone)
        if err != nil {
                panic(err)
        }
        // Mix: left hemi StepBP, right hemi StepTweenChain
        hemi := s.Children[1].(*parallel.Layer)
        hemi.SetBranchModes(parallel.ModeStepBP, parallel.ModeStepTweenChain)

        x := core.NewTensor[float32](1, 8)
        t := core.NewTensor[float32](1, 1)
        t.Data[0] = 0.5
        loss, err := parallel.TrainStackMSE(s, x, t, parallel.ModeStepBP, 0.01)
        fmt.Println("loss", loss, "err", err)
}
```

Output

exit 0 · last run via `go run .`

```
loss 0 err <nil>
```

CHAPTER 28

layers/metacognition

github.com/openfluke/welvet/layers/metacognition

Why it exists

Observed layers can apply heuristic stability rules (gate/scale/reset) without dtype morph/QAT.

What it is

Wraps Dense + DefaultStabilityRules(); Stats exposed. Full timed matrix + train grids.

Go example

examples/28-metacognition/main.go

Run: `cd welvet/examples/28-metacognition && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/metacognition"
)

func main() {
	l, err := metacognition.New(metacognition.Config{
		Dim: 16, Rules: metacognition.DefaultStabilityRules(),
	})
	if err != nil {
		panic(err)
	}
	x := core.NewTensor[float32](1, 16)
	_, y, err := metacognition.Forward(l, x)
	fmt.Println(len(y.Data), err)
}
```

Output

exit 0 · last run via `go run .`

```
16 <nil>
```

CHAPTER 29

runtime/forward

github.com/openfluke/welvet/runtime/forward ✅

Why it exists

A grid of heterogeneous ops needs one walker that dispatches by concrete type and fails loudly on unknowns.

What it is

Forward[T](grid, input) → Result tape; Cell[T] for single-cell dispatch. Covers Dense… Residual + extended wired ops.

Go example

examples/29-forward/main.go

Run: `cd welvet/examples/29-forward && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/architecture"
	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/dense"
	"github.com/openfluke/welvet/runtime/forward"
)

func main() {
	g := architecture.NewGrid(1, 1, 1, 1)
	l, _ := dense.New(8, 8, core.ActivationLinear, core.DTypeFloat32)
	_ = dense.Place(g, 0, 0, 0, 0, l)
	res, err := forward.Forward(g, core.NewTensor[float32](1, 8))
	fmt.Println(res != nil, err)
}
```

Output

exit 0 · last run via `go run .`

```
true <nil>
```

CHAPTER 30

runtime/backward

github.com/openfluke/welvet/runtime/backward

Why it exists

Training needs a reverse tape over the same ops forward used — no separate graph framework.

What it is

Backward[T](fwdResult, gradOut) walks the tape and returns per-op weight grads.

Go example

examples/30-backward/main.go

Run: `cd welvet/examples/30-backward && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/architecture"
	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/dense"
	"github.com/openfluke/welvet/runtime/backward"
	"github.com/openfluke/welvet/runtime/forward"
)

func main() {
	g := architecture.NewGrid(1, 1, 1, 1)
	l, _ := dense.New(4, 4, core.ActivationLinear, core.DTypeFloat32)
	_ = dense.Place(g, 0, 0, 0, 0, l)
	fwd, _ := forward.Forward(g, core.NewTensor[float32](1, 4))
	bwd, err := backward.Backward(fwd, core.NewTensor[float32](1, 4))
	fmt.Println(bwd != nil, err)
}
```

Output

exit 0 · last run via `go run .`

```
true <nil>
```

CHAPTER 31

runtime/training

github.com/openfluke/welvet/runtime/training ✅

Why it exists

Suites and small nets need MSE+SGD and tween hooks without inventing an external trainer or a retained float32 master beside storage.

What it is

MSE/MSEGrad, SGD, Step, ApplyTween/StepTween, StepMesh. Layer-agnostic ApplyGradSGD dispatch (Dense…Mamba/GDN/…). FormatNone: in-dtype ApplySGD; packed: unpack→update→re-Pack. No QAT dual path — storage dtype/format is truth after every step. Sandwich credit (Split / FastProxy / Sparse / …) lives in layers/parallel TrainMode — see §67.

Grid `training.Step` is MSE + backprop on a volumetric tape. Cameral Mix and the 29 named credit updates use `parallel.TrainStackMSE` (and Mesh* schedulers) so hemispheres can disagree. Equations, rival metric (hard Acc vs StepBP), and Lucy Score honesty: §67.

Go example

examples/31-training/main.go

Run: `cd welvet/examples/31-training && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/architecture"
        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/dense"
        "github.com/openfluke/welvet/runtime/forward"
        "github.com/openfluke/welvet/runtime/training"
```

```
)

func main() {
        g := architecture.NewGrid(1, 1, 1, 1)
        l, _ := dense.New(4, 4, core.ActivationLinear, core.DTypeFloat32)
        _ = dense.Place(g, 0, 0, 0, 0, l)
        fwd, _ := forward.Forward(g, core.NewTensor[float32](1, 4))
        loss, err := training.Step(fwd, core.NewTensor[float32](1, 4), 1e-2)
        fmt.Println(loss, err)
}
```

Output

exit 0 · last run via `go run .`

```
0 <nil>
```

CHAPTER 32

runtime/step — step mesh

github.com/openfluke/welvet/runtime/step ✅

Why it exists

Spatial feedback (remote links) needs a discrete-time mesh where every cell updates from a double buffer — different from a decoder wavefront. Cross-numeric train also needs the same mesh with weight DType ⊥ activation Tensor[T].

What it is

State[T], StepForward/StepBackward/StepApplyTween / StepMesh across the grid for all wired Ops × dtype × quant × CPU/SIMD. W2A Cross-Numeric Train: polyops.AllKinds() × weight dtype × act host (smoke ~21×7×5 ≈ 735; full ~21×34×15 ≈ 10.7k) — asserts no retained f32 master after StepMesh. Stack Step* is a different clock: IsLineStep / TrainLine (1D pipe) — see §67.

Two clocks named Step

This package is the volumetric mesh: every grid cell hops from a double buffer. MeshBP / MeshTween / MeshTweenChain call it.

Stack Step* (`IsLineStep`) is a 1D systolic pipe on a Sandwich — not this mesh. One sample enters child 0 per `TrainStackMSE` tick; every in-flight sample advances one layer; the output/train event is the sample that entered *depth* ticks ago. Fill ticks do not update. Serve stays a full `ForwardStack` . Mesh* still needs a Grid. Named updates: §67.

Go example

examples/32-step/main.go

Run: `cd welvet/examples/32-step && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/architecture"
        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/dense"
        "github.com/openfluke/welvet/runtime/step"
)

func main() {
        g := architecture.NewGrid(1, 1, 1, 1)
        l, _ := dense.New(4, 4, core.ActivationLinear, core.DTypeFloat32)
        _ = dense.Place(g, 0, 0, 0, 0, l)
        st := step.New[float32](g)
        _, err := step.StepForward(g, st, false)
        fmt.Println(err)
}
```

Output

exit 0 · last run via `go run .`

```
<nil>
```

CHAPTER 65

Cross-numeric train + down-the-dem

github.com/openfluke/welvet/runtime/training ✅

Why it exists

Weight storage dtype and activation Tensor[T] are independent axes. Proving train without a retained float32 master means sweeping W×A — not only matched float32 acts.

What it is

W2A Step Cross-Numeric Train: polyops.AllKinds() × FormatNone weight dtype × Go Numeric act host (smoke ~735; full ~10.7k) via StepMesh, then assert no retained f32 master. Public Dense volumetric showcase: down-the-dem — dtype demotion ladder, packed quants, and the full 34×15×3 perm matrix with charts/PDF.

down-the-dem — same 5-layer Dense stack on cubes 1^3 / 2^3 / 3^3; only the number story changes:

- **dtype** — FormatNone demotion (wide → 1-bit), in-dtype SGD
- **quant** — packed formats; unpack → update → re-Pack
- **perm** — every weight dtype × every act host (1,530 cells on three grids)

Repo + report: github.com/openfluke/down-the-dem · PDF: `report/down-the-dem.pdf` in that repo. Honesty: runnable storage-truth train ≠ task accuracy or native low-precision GEMV ALU.

Go example

examples/65-cross-numeric/main.go

Run: `cd welvet/examples/65-cross-numeric && source ../env.sh && go run .`

```
package main

import (
```

```
        "fmt"

        "github.com/openfluke/welvet/architecture"
        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/dense"
        "github.com/openfluke/welvet/quant"
        "github.com/openfluke/welvet/runtime/training"
)

func main() {
        // Weight storage: int8 FormatNone. Activations/grads: Tensor[int8].
        g := architecture.NewGrid(1, 1, 1, 1)
        init := make([]float32, 4*4)
        for i := range init {
                init[i] = 0.1
        }
        l, err := dense.NewConfigured(4, 4, core.ActivationLinear, core.DTypeInt8,
quant.FormatNone, init)
        if err != nil {
                panic(err)
        }
        if err := dense.Place(g, 0, 0, 0, 0, l); err != nil {
                panic(err)
        }
        x := core.NewTensor[int8](1, 4)
        y := core.NewTensor[int8](1, 4)
        for i := 0; i < 4; i++ {
                x.Data[i] = int8(i + 1)
                y.Data[i] = int8(i)
        }
        loss, _, err := training.StepMesh(g, x, y, 1, 0.05)
        fmt.Println(loss, err, l.Weights.RetainsF32Master())
}
```

Output

exit 0 · last run via `go run .`

```
1.5 <nil> false
```

CHAPTER 67

TrainMode — 29 named updates

github.com/openfluke/welvet/layers/parallel ✅ 29 modes

Why it exists

Backprop is one update, not the only one. Credit assignment (broadcast gap, head proxy, sparse duty clock) has to be a named axis you can race — not a comment in a notebook. Cameral Mix also needs one TrainMode per hemisphere on the same loss.

What it is

parallel.TrainMode: AllNamedTrainModes() = 29 (Inherit omitted). Stack-local Split/Alt plus Step* 1D pipe twins and Mesh* grid schedulers. TrainStackMSE / TrainStackCE honour BranchModes. Display names use Short() / ShortTrainMode. Rival metric is hard Acc vs StepBP; Lucy Score is Tput × Avail × Acc — do not mix those sentences.

> **Honesty**
> Rival = hard Acc vs StepBP. Lucy Score rewards skip-GEMV (Sparse Avail). TweenChain on a Sandwich is chain-rule BP under another name. Mesh* on an origin-only cube collapses to the family (not 8/27 trained copies). FastProxy is DFA with B := W_head^T, not a learned random B. Do not write “Sparse beat backprop.”

Loss gap (MSE or CE)

```
MSE:  L = (1/d) ||ŷ − t||²     g_y = (2/d)(ŷ − t)
CE:   L = −mean log softmax(ŷ)_class     g_y = (p − t) / B
```

Head always sees `g_y`. Classification hosts call `TrainStackCE` so Acc can leave chance; MSE on a one-hot stays uniform. What each mode does with `g_y` is the whole story.

Families (29 named tokens)

Family	Tokens	Update
Backprop	NormalBP · StepBP · MeshBP	chain rule J^T through the tape / volumetric Step
Tween	Tween · StepTween · MeshTween	broadcast P(g_y) onto every leaf; η ← η/2
TweenChain	TweenChain · StepTweenChain · MeshTweenChain	same math as BP on a Sandwich
Split	TweenSplit · StepTweenSplit · MeshTweenSplit	g_i = (1/N) P(g_y)
Alt	TweenAlt · StepTweenAlt · MeshTweenAlt	Split then re-forward then Tween (half LR)
HeadProxy	TweenSplitHeadProxy · StepTweenSplitHeadProxy	head J^T g_y (with act′); hidden dW only
FastProxy	TweenSplitFastProxy · StepTweenSplitFastProxy · MeshTweenSplitFastProxy	g_proxy = W_head^T g_y (skip act′)
Linear	TweenSplitLinear · StepTweenSplitLinear	affine W^T walk; skip ⊙ act′; hemispheres share the down-vector
LinearCache	TweenSplitLinearCache · StepTweenSplitLinearCache	cache every 20 steps; dead on sine — control
HeadProxyAsync	TweenSplitHeadProxyAsync · StepTweenSplitHeadProxyAsync	hidden uses proxy from T−1; not EMA

Family	Tokens	Update
Sparse	TweenSplitSparse · StepTweenSplitSparse · MeshTweenSplitSparse	head + one rotating hidden; other dW = 0

`AllCreditTrainModes()` = 16 stack-local Split/Alt plus Step* credit twins (no Mesh). `AllMeshCreditTrainModes()` = 4 Mesh credit (no HeadProxy / Linear / LinearCache / HeadProxyAsync Mesh twins). `AllStackLocalTrainModes()` = 22 (no Inherit, no Mesh*). `AllNamedTrainModes()` = 29 — the Test49 / test50 set. `IsLineStep()` = 11 (StepBP, StepTween, StepTweenChain, StepTweenSplit, StepTweenAlt, plus the six Step* credit twins).

Tables and logs use `Short()` / `ShortTrainMode` (legend: `[T]=Tween [S]=Split [FP]=FastProxy [L]=Linear [HP]=HeadProxy`). Persistence and `ParseTrainMode` still use the full `String()` token. New Step* iota values were appended so old uint8 numbers stay stable.

Equations

Backprop (the rival)

```
head:   g_head = J_head^T g_y
hemi:   g_hemi = J_hemi^T g_head
stem:   g_stem = J_stem^T g_hemi
dW_i from local Backward(g_i, x_i)
```

SIMD GEMV. This is who FastProxy has to beat on hard Acc — not Lucy Score.

Tween — broadcast, half LR

```
g_i = P(g_y)     η ← η/2
dW_i = localBackward(g_i, x_i)
```

Blind to W^T sign. Sine Acc collapses. MeshTween is the volumetric scheduler of this family, not a secret FastProxy.

TweenSplit — even split

```
g_i = (1/N) P(g_y)
```

Still not J^T. Cheap Acc ceiling. Score can rise because the update is cheap (Avail), not because g is better.

TweenAlt — Split then Tween

Per sample, AltTimes times (default 1): Split from live g_y → re-forward → Tween from g_y′ (half LR). Extra forwards kill Avail → Score last.

HeadProxy

```
g_proxy = J_head^T g_y = W_head^T (g_y ⊙ act′(pre_head))
hidden i = 1…N−1:  g_i = 1/(N−1) P(g_proxy)
                   dW_i = g_i x_i^T     (no discarded W^T)
```

One real J_head^T. Hemispheres do *not* get J_hemi^T.

FastProxy

```
g_proxy = W_head^T g_y          ← skip act′
head dW still uses act′
hidden: same 1/(N−1) P(g_proxy) dW-only as HeadProxy
```

DFA with B := W_head^T, not a learned random B. On AAI test50 sine, FastProxy SoftAcc often sits above StepBP while both are at 100% hard Acc — that is the FastProxy vs BP sentence.

Linear

```
g_head↓  = g_y
g_hemi↓  = W_head^T g_y          (siblings share the vector)
g_stem↓  = Σ_hemi W_hemi^T g_hemi↓
then every leaf: g_i = (1/N) P(g_i↓),  dW_i = g_i x_i^T
```

Same $O(N^2)$ class as backprop. Score stays StepBP-class unless Acc is way up.

LinearCache — dead control

```
every 20 steps: full Linear walk, cache g_i↓
else: g_i ← g_i^cache · ||g_y||_live / ||g_y||_cache
```

Norm scaling cannot recover sign flips after a frequency switch. If this wins sine, the board is lying.

HeadProxyAsync

```
g_hidden^(T) = 1/(N-1) P(g_proxy^(T-1))
head computes g_proxy^(T) = W_head^T g_y^(T) for next step
```

First sample seeds live. Stale sign on XOR. Not EMA.

Sparse — duty clock

```
g_proxy = W_head^T g_y
dW_head = g_y x_head^T
k = t mod (N-1)
dW_k    = P(g_proxy) x_k^T
other leaves: dW = 0 this sample
```

Real FLOP cut → Avail 40–50% → Lucy Score explodes. That is a duty clock, not a smaller big-O than backprop and not a better chain rule. On test50 copy, Sparse often *loses* Acc vs StepBP while winning Score.

Step* — 1D systolic pipe

The Step prefix on a stack mode is a *schedule*, not a second leftover-forward pass and not Mesh*. `IsLineStep` → `TrainLine` / `trainStackLine`: one sample enters child 0 per tick; every in-flight sample advances one layer; the output (and the train event) is the sample that entered *D* ticks ago. Fill ticks do not update. Serve stays a full `ForwardStack`.

Same family update on the Sandwich whether Step or not — HeadProxy / Linear / FastProxy / Sparse / Async credit walk `trainTweenSplitLeaves` on both twins. Mesh* still requires a Grid (`RequiresGrid`). There is no Mesh HeadProxy / Linear / LinearCache / HeadProxyAsync.

Mesh*

MeshBP = volumetric `training.Step`. MeshTween = `StepMesh`. MeshTweenChain = `StepTween`. Mesh Split / Alt / FastProxy / Sparse credit the placed stack under a grid walk. On origin-only $1^3/2^3/3^3$ (rest `IsDisabled`) Mesh* usually matches the stack twin. Cube size is hop topology, not 27 sandwiches.

Where it is raced

- **w2a Test49** — permutation smoke: 29 modes × $1^3/2^3/3^3$ × Parallel / Bicameral / poly kinds, origin-only. In `[0] Run ALL`. Not a Lucy race.

- **AAI test48** — credit sweep (layers × dtypes × short jobs) with these equations.
- **AAI test50** — FP32 Lucy race, all 29, cams 1–3, cubes 1–3. Copy: Split/Alt family +5 to +12 Acc vs StepBP. Sine: Acc ceiling; FastProxy SoftAcc is the knife. XOR: 4-point parking lot (75%). Sparse wins Score, not Acc.

Cameral why + sandwich stem→mid→head: §68. Measuring math: §66.

Go example

examples/67-train-modes/main.go

Run: `cd welvet/examples/67-train-modes && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/layers/parallel"
)

func main() {
        named := parallel.AllNamedTrainModes()
        line := 0
        for _, m := range named {
                if m.IsLineStep() {
                        line++
                }
        }
        fp, err := parallel.ParseTrainMode("stepfastproxy")
        fmt.Println("named", len(named), "linestep", line)
        fmt.Println("stepfastproxy", fp.Short(), err)
        fmt.Println(parallel.ShortTrainModeLegend)
}
```

Output

exit 0 · last run via `go run .`

```
named 31 linestep 11
stepfastproxy Step[T][S][FP] <nil>
[T]=Tween  [S]=Split  [FP]=FastProxy  [L]=Linear  [HP]=HeadProxy  [F]=Freeze
[Sh]=Shadow  [A]=Adv  [M]=Memory
```

CHAPTER 33

systems/dna

github.com/openfluke/welvet/systems/dna ✅

Why it exists

Quant and train must be measurable as topology/weight fingerprints — DNA detects logic shifts.

What it is

ExtractDNA, CompareNetworks, CosineSimilarity, FlattenOp / CollectStores across dtype×quant.

Go example

examples/33-dna/main.go

Run: `cd welvet/examples/33-dna && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/architecture"
	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/dense"
	"github.com/openfluke/welvet/systems/dna"
)

func main() {
	g := architecture.NewGrid(1, 1, 1, 1)
	l, _ := dense.New(4, 4, core.ActivationLinear, core.DTypeFloat32)
	_ = dense.Place(g, 0, 0, 0, 0, l)
	a := dna.ExtractDNA(g)
	fmt.Println(dna.CompareNetworks(a, a))
}
```

Output

exit 0 · last run via `go run .`

```
{1 map[0,0,0,0:1] []}
```

CHAPTER 34

systems/evolution

github.com/openfluke/welvet/systems/evolution

Why it exists

Topology search and weight crossover need first-class splice + NEAT on CPU-resident grids.

What it is

SpliceDNA, NEATMutate, NewNEATPopulation, CloneGrid — dtype/quant preserved via SetFromF32.

Go example

examples/34-evolution/main.go

Run: `cd welvet/examples/34-evolution && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/architecture"
	"github.com/openfluke/welvet/systems/evolution"
)

func main() {
	a, b := architecture.NewGrid(1, 1, 1, 1), architecture.NewGrid(1, 1, 1, 1)
	child, err := evolution.SpliceDNA(a, b, evolution.DefaultSpliceConfig())
	if err != nil {
		fmt.Println(err)
		return
	}
	_, err = evolution.NEATMutate(child, evolution.DefaultNEATConfig(16))
	fmt.Println(err)
}
```

Output

exit 0 · last run via `go run .`

```
<nil>
```

CHAPTER 35

systems/tween

github.com/openfluke/welvet/systems/tween

Why it exists

Target propagation (chain-rule or Hebbian layerwise gaps) is an alternative credit-assignment path. Not the same package as TrainMode Tween / TweenChain on a Sandwich (those are layers/parallel — §67).

What it is

NewState, Forward, BackwardChainRule / BackwardLayerwise, ApplyGaps; SIMD DotTile/Saxpy budgets.

Go example

examples/35-tween/main.go

Run: `cd welvet/examples/35-tween && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/architecture"
        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/systems/tween"
)

func main() {
        g := architecture.NewGrid(1, 1, 1, 1)
        st := tween.NewState[float32](g, tween.DefaultConfig())
        _, err := tween.Forward(g, st, core.NewTensor[float32](1, 4))
        fmt.Println(err)
}
```

Output

exit 0 · last run via `go run .`

```
forward: no op at {0 0 0 0} (type Dense)
```

CHAPTER 36

systems/tanhi — TANHI · UDP HUD

github.com/openfluke/welvet/systems/tanhi

Why it exists

Training visualization must never block the engine — best-effort UDP JSON-lines to a HUD.

What it is

TANHI = *Tensor Activation Network Holographic Interface*. Sparse non-blocking JSON-line UDP events for per-layer forward/backward HUD visualization (timing, dtypes, shapes, routing links, parallel branches/combine). **v1.1.1:** grid BP, cameral stack/parallel train (all mode families), step mesh; `ConfigureGrid`, `SetTanhi`, topology bridge. `ConfigFromGrid`, `Emit` / `EmitSweep`, `DefaultUDPPort` (17481). SoulGlitch-style consumers.

Go example

examples/36-tanhi/main.go

Run: `cd welvet/examples/36-tanhi && source ../env.sh && go run .`

```
package main

import (
        "github.com/openfluke/welvet/architecture"
        "github.com/openfluke/welvet/systems/tanhi"
)

func main() {
        g := architecture.NewGrid(1, 1, 1, 1)
        cfg := tanhi.ConfigFromGrid(g)
        tanhi.EmitSweep(cfg, "epoch-0") // non-blocking best-effort
}
```

Output

exit 0 · last run via `go run .`

```
(no output)
```

CHAPTER 37

systems/telemetry

github.com/openfluke/welvet/systems/telemetry

Why it exists

Static structural blueprints (sizes, op kinds) differ from live TANHI events.

What it is

ExtractNetworkBlueprint, ExtractLayerTelemetry for introspection/UIs.

Go example

examples/37-telemetry/main.go

Run: `cd welvet/examples/37-telemetry && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/architecture"
        "github.com/openfluke/welvet/systems/telemetry"
)

func main() {
        g := architecture.NewGrid(1, 1, 1, 1)
        bp := telemetry.ExtractNetworkBlueprint(g, "demo")
        fmt.Printf("%+v\n", bp)
}
```

Output

exit 0 · last run via `go run .`

```
{ID:demo TotalLayers:1 TotalParams:0 Layers:[{Z:0 Y:0 X:0 L:0 Type:Dense
```

```
Activation:ReLU Parameters:0 InputShape:[] OutputShape:[] Branches:[] CombineMode:}]}
```

CHAPTER 66

lucy — SoftAcc / Score measuring

github.com/openfluke/welvet/lucy

Why it exists

Adaptation benches (test41-w, tide, live_gpt) need one shared measuring math — SoftAcc, Availability, AdaptPct, Score — not three copies of the formulas.

What it is

Pure measuring package: SoftAcc / SoftAccProb, Window + Snapshot, Finalize. No datasets, no train loops. Sine scale 0.10; classification SoftAccProb scale 1.0. Density / synthetic-organism board is §69.

```
SoftAcc      = 100 × (1 − |pred−target| / scale)   clamped [0,100]
Availability = InferMs / (InferMs + TrainMs) × 100
Score        = Throughput × Availability × Acc / 10_000
Acc          = hard argmax % (AvgAccuracy) — the Acc pillar
AdaptPct     = mean SoftAcc in AdaptWindows after each switch
```

Lucy Score terms. Acc is argmax. SoftAcc is serve-confidence, not Score.

`tide/metrics` re-exports this package for existing tide callers. Engine tests for lucy live under `w2a/tests/lucy`.

> **Acc ≠ Score**
> Hard Acc is the rival vs StepBP. Score = Throughput × Availability × *Acc* / 10_000. Sparse can win Score (skip-GEMV Avail) and lose Acc. AAI test50 copy: Split family +5 to +12 Acc vs StepBP; Sparse Score ~8k–10k while Acc is often worse than StepBP. Sine: many modes sit at 100% hard Acc; FastProxy SoftAcc is the knife. XOR is a 4-point parking lot (75%) — smoke, not a ranking. Consciousness / density: §69.

AAI Lucy benches (private tree, `replace` → public Welvet): test41 native cam, test48 credit sweep, test50 23-mode FP32 race. Harness is not an engine package. Pulse math is this chapter. Density board: §69. Modes + equations: §67. Cameral sandwiches: §68.

Go example

examples/66-lucy/main.go

Run: `cd welvet/examples/66-lucy && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/lucy"
)

func main() {
        a := lucy.SoftAccOne(0.72, 0.80) // sine scale 0.10
        p := lucy.SoftAccProb(0.91, 1.0) // class scale 1.0
        var snap lucy.Snapshot
        snap.AvgAccuracy = 80
        snap.SoftAcc = a
        snap.InferMs = 8
        snap.TrainMs = 2
        snap.Throughput = 12000
        lucy.Finalize(&snap, lucy.Options{AdaptWindows: 10})
        fmt.Printf("soft=%.1f class=%.1f avail=%.1f score=%.0f\n",
                a, p, snap.Availability, snap.Score)
}
```

Output

exit 0 · last run via `go run .`

```
soft=20.0 class=91.0 avail=80.0 score=7680
```

CHAPTER 69

Lucy density — synthetic organism

github.com/openfluke/welvet/lucy ✅ BuildLPD

Why it exists

A new host (char LM, MNIST, a layer sprint) should not copy tide's goldilocks math. The question is always the same: can the net *run and train at the same time* in a small box, then how far that live-fit condenses without becoming a trap. That ruler belongs in the engine.

What it is

lucy.BuildLPD ranks Samples for consciousness (Acc / Throughput / Availability keep vs learner peaks) then Lucy density (Q × shrink vs Acc-champ RAM). Gold / near / keep / trap bands, consciousness radar, and memory-density radar are computed here. Tide dash and Lucy PDF only draw the board.

```
WHY
  Industry ships dead calculators: INT8 inference, no on-device train.
  Synthetic organism = serve while learning, then shrink without falling
  into chance-Acc "tiny & fast" traps.

PILLARS (consciousness)
  Acc    = argmax %          (did it learn?)
  Thru   = outputs / second  (can it act while live?)
  Avail  = InferMs / busy    (can you still talk to it while it trains?)
  Score  = T × Avail × Acc / 10_000     live-fit
  SoftAcc is serve-confidence — not a pillar.

DENSITY (memory intelligence)
  Acc champ = RAM reference (best learner, not the Score champ)
  Learner   = RelAcc ≥ 70% of Acc champ   (traps do not set Thru/Avail peaks)
  Q         = geomean(RelAcc, RelThru, RelAvail)
  shrink    = AccChampRAM / thisRAM   (capped ×32)
  LPD       = Q × shrink    if RelAcc ≥ 70% else 0

BANDS
  gold  all 3 pillars ≥80% and RAM ≤20% of Acc champ
  near  Acc + (Thru or Avail) ≥80% at ≤50% RAM
  trap  RAM ≤20% and Acc keep <70%     (binary looking dense)
  Score/MiB is the trap metric — do not use it for goldilocks.

RADARS (for PDF / dash — math here, drawing in tide)
```

```
Consciousness  = (RelAcc, RelThru, RelAvail)
Memory density = (RelAcc, RelThru, RelAvail) × shrink   traps at origin
```

One ruler. Any host feeds Samples; tide is just the display.

Why we measure this

The goal is not a better chatbot and not a paper for a faculty board. It is a **synthetic organism**: a net that owns time — continuous serve, continuous train, no turn-based halt, no phone-home. Lucy Score asks the thermodynamic question tide was built for: *does SGD that blocks inference die, and do proxy / Split / Sparse paths keep the live loop while still learning?*

Hard Acc is whether it learned. Availability is whether it still breathes. Throughput is how many actions it took while both were happening. Multiply them and you get live-fit. SoftAcc is how confident the serve looked — useful, not the Acc term.

Memory density asks the second question: once you have a learner, how far can dtype / quant / cameral width *condense* that live-fit versus the Acc champ's RAM without collapsing to chance Acc. A binary cell that is 30× smaller and 4× faster with 12% Acc is a **trap** (LPD = 0). An int8 that keeps ≥70% of Acc-champ Acc at a fifth the RAM is goldilocks.

Formulas

Symbol	Formula	Meaning
Availability	`InferMs / (InferMs + TrainMs) × 100`	Duty cycle. SGD that blocks serve dies here.
Throughput T	`TotalOutputs / duration_s`	Actions per second while the sweep is live.
Acc	argmax % (`AvgAccuracy`)	Learning. Acc champ is the RAM reference.
Lucy Score	`T × Avail × Acc / 10_000`	Live-fit. SoftAcc is not this term.
Realtime	`T × Avail / 100`	Duty-cycle speed without Acc.

Symbol	Formula	Meaning
ZeroDowntime	`Acc × Avail / 100`	Still serving while learning.
RelAcc / RelThru / RelAvail	value / learner peak, clamped [0,1]	Keep vs cells that actually learn.
Q	geomean of the three Rel*	Consciousness mix. Traps do not set Thru/Avail peaks.
shrink	`min(AccChampRAM / thisRAM, 32)`	How much smaller than the Acc champ.
LPD	`Q × shrink` if RelAcc ≥ 0.70 else 0	Lucy density. Weeds Score/MiB traps.
Gold	all 3 Rel* ≥ 0.80 and RAM ≤ 20% Acc champ	Trifecta in a small box.
Gold-std	Acc ≥ 80% plus Thru or Avail, then smallest then fastest	Two-or-more of the trifecta.
Trap	RAM ≤ 20% Acc champ and RelAcc < 0.70	Tiny and fast with chance Acc.
MobileScore	`Score / WeightMiB`	The binary trap. Use LPD instead.

What tide still owns

Tide is the *race*: serve+train pulses, permute matrix, dashboard, Lucy PDF (including drawing the two radars). `tide/report.BuildLPD` pretty-prints cell IDs then calls `lucy.BuildLPD`. A new thing that only needs the ruler:

```
import "github.com/openfluke/welvet/lucy"
board := lucy.BuildLPD(samples)
_ = board.Top[0].Consciousness()  // Acc, Thru, Avail keep
_ = board.Top[0].MemoryDensity()  // same × shrink; traps at 0
```

Classification hosts should train with `TrainStackCE` so the Acc pillar can leave chance; MSE on a one-hot stays uniform. That is a loss gap in `layers/parallel`, not a Lucy formula.

Pulse math: §66. Train modes: §67.

Go example

examples/69-lucy-density/main.go

Run: `cd welvet/examples/69-lucy-density && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/lucy"
)

func main() {
        board := lucy.BuildLPD([]lucy.Sample{
                {ID: "f32", Mode: "sgd", Acc: 90, Thru: 200, Avail: 40, Score: 100,
RAMKiB: 1000},
                {ID: "int8", Mode: "sgd", Acc: 82, Thru: 180, Avail: 38, Score: 85,
RAMKiB: 180},
                {ID: "bin", Mode: "sgd", Acc: 12, Thru: 400, Avail: 50, Score: 40,
RAMKiB: 40},
        })
        top := board.Top[0]
        fmt.Printf("lead=%s band=%s LPD=%.2f trap=%s\n",
                top.ID, top.Band, top.LPD, board.Trap[0].ID)
}
```

Output

exit 0 · last run via `go run .`

```
lead=int8 band=gold LPD=5.11 trap=bin
```

CHAPTER 38

model/entity — .entity files

github.com/openfluke/welvet/model/entity

Why it exists

HF safetensors are awkward for native topology + packed weights. ENTITY is the Welvet checkpoint.

What it is

Open/Inspect/IsEntity, LoadBlob/LoadQuantBlob, PackFromHF/ImportFromHF, WriteTransformerFile, SerializeNetwork, WriteCameralFile / LoadCameral (sandwich Stack + TrainMode).

Go example

examples/38-entity/main.go

Run: `cd welvet/examples/38-entity && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/model/entity"
)

func main() {
        info, err := entity.Inspect("model.entity")
        if err != nil {
                fmt.Println(err)
                return
        }
        ef, err := entity.Open("model.entity")
        if err != nil {
                panic(err)
        }
        defer ef.Close()
```

```
        fmt.Println(info, ef.HasTokenizerBlob())
}
```

Output

exit 0 · last run via `go run .`

```
open model.entity: no such file or directory
```

CHAPTER 39

model/hf — snapshots

github.com/openfluke/welvet/model/hf

Why it exists

Import starts with probing HF/MLX layouts before packing ENTITY.

What it is

InspectSnapshot, DetectArchitecture, safetensors/MLX loaders, Qwen3.5 hybrid helpers.

Go example

examples/39-hf/main.go

Run: `cd welvet/examples/39-hf && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/model/hf"
)

func main() {
        info, err := hf.InspectSnapshot("/path/to/hf-snapshot")
        if err != nil {
                fmt.Println(err)
                return
        }
        fmt.Println(info)
}
```

Output

exit 0 · last run via `go run .`

```
hf: config.json: stat /path/to/hf-snapshot/config.json: no such file or directory
```

CHAPTER 40

model/tokenizer

github.com/openfluke/welvet/model/tokenizer ✅

Why it exists

Generate needs encode/decode of HF tokenizer.json without pulling Python.

What it is

LoadTokenizer, Encode/Decode, LoadForEntity.

Go example

examples/40-tokenizer/main.go

Run: `cd welvet/examples/40-tokenizer && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/model/tokenizer"
)

func main() {
        tok, err := tokenizer.LoadTokenizer("tokenizer.json")
        if err != nil {
                fmt.Println(err)
                return
        }
        ids := tok.Encode("hello welvet", true)
        fmt.Println(ids, tok.Decode(ids, true))
}
```

Output

exit 0 · last run via `go run .`

```
failed to read tokenizer file: open tokenizer.json: no such file or directory
```

CHAPTER 41

model/sampling

github.com/openfluke/welvet/model/sampling

Why it exists

Logits → token ID needs ArgMax, TopK+temperature, penalties, and chat hygiene in one place.

What it is

ArgMax, SampleTopK, ApplyRepetitionPenalty, BanIDs, SanitizeChatReply.

Go example

examples/41-sampling/main.go

Run: `cd welvet/examples/41-sampling && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/model/sampling"
)

func main() {
        logits := []float32{0.1, 2.4, 0.3, -1}
        fmt.Println(sampling.ArgMax(logits))
        fmt.Println(sampling.SampleTopK(logits, 2, 0.8, true))
}
```

Output

exit 0 · last run via `go run .`

```
1
```

```
1
```

CHAPTER 42

model/transformer — generate

github.com/openfluke/welvet/model/transformer

Why it exists

ENTITY packs must run as Llama-style decoders with KV cache, profiles (SIMD/WebGPU/fused), and chat templates.

What it is

LoadEntity → Model; Generate; ApplyExec profiles; ExportFusedGPUSpec / hybrid sync.

Go example

examples/42-transformer/main.go

Run: `cd welvet/examples/42-transformer && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/model/tokenizer"
        "github.com/openfluke/welvet/model/transformer"
)

func main() {
        m, err := transformer.LoadEntity("model.entity")
        if err != nil {
                fmt.Println(err)
                return
        }
        tok, err := tokenizer.LoadTokenizer("tokenizer.json")
        if err != nil {
                panic(err)
        }
        if err := m.ApplyExec(transformer.ProfileSIMDMultiCore()); err != nil {
                panic(err)
        }
        text, _, err := m.Generate(
```

```
                tok.Encode, tok.Decode, nil,
                "You are helpful.", "Say hi.",
                transformer.GenOptions{MaxTokens: 32, Silent: true},
        )
        fmt.Println(text, err)
}
```

Output

exit 0 · last run via `go run .`

```
open model.entity: no such file or directory
```

CHAPTER 43

apps — octo · flux2 · mosstts

github.com/openfluke/welvet/apps/...

Why it exists

Products must not pollute engine packages. Octo is the model shell; flux2/mosstts are domain apps. Lucy *races* (AAI test41 / test48 / test50) are benches, not Welvet packages.

What it is

octo (own module): download/convert/chat, see §44. flux2: MMDiT image. mosstts: Speak/SpeakToFile pipeline. AAI sandwiches race TrainMode on toys — see §68. Off the v1.0 engine board.

The engine never imports apps. AAI is a separate Lucy harness that `replace`s `github.com/openfluke/welvet`. It is how we talk about cameral Mix and credit modes with numbers (hard Acc, SoftAcc, Score) instead of slogans. Public API stays in `layers/parallel` + `lucy`.

Go example

examples/43-apps/main.go

Run: `cd welvet/examples/43-apps && source ../env.sh && go run .`

```
package main

import "fmt"

func main() {
        fmt.Println("Apps import the engine — the engine never imports apps.")
        fmt.Println("  cd apps/octo && go run .")
        fmt.Println("  cd apps/flux2 && go run .")
        fmt.Println("  cd apps/mosstts && go run .")
}
```

Output

exit 0 · last run via `go run .`

```
Apps import the engine — the engine never imports apps.
  cd apps/octo && go run .
  cd apps/flux2 && go run .
  cd apps/mosstts && go run .
```

Validate (harness)

```
cd apps/octo && go run .   # when module wired
```

CHAPTER 44

Octo — model shell

github.com/openfluke/welvet/apps/octo ✅ runs

Why it exists

A model is only useful with a shell around it: pull weights from Hugging Face, convert them to a Welvet `.entity`, then chat, serve, or benchmark. Octo is that shell, kept in its own module so the engine never depends on an app.

What it is

Subcommands cover the whole loop: `hub` download/ensure repos, `convert` pack to a single `.entity`, interactive `run` / `serve` /chat, plus image and speech menus. Its `bench` harness sweeps every quant format across a CPU Plan 9 SIMD fused profile and a WebGPU fused profile.

Bench: SmolLM2-135M-Instruct across quant formats

template smol2-135m-fuse · linux/amd64 · NVIDIA GTX 1650 SUPER · 3m17s · 80 runs (76 ok, 4 skipped: "none" has no fused kernel). Throughput is total tokens/s on the first prompt; entity is on-disk size.

Quant	GPU-fused tok/s	SIMD-fused tok/s	Entity MB
Q8_0	108.2	33.2	252.5
Q6_K	73.5	4.9	216.4
Q5_K	71.7	6.3	208.4
Q4_K	73.4	7.8	192.3
Q4_0	112.1	36.8	188.3
IQ4_XS	105.5	7.6	204.4
IQ4_NL	84.1	8.8	188.3

Quant	GPU-fused tok/s	SIMD-fused tok/s	Entity MB
TernaryPacked	106.9	18.9	172.3
BinaryPacked	88.3	21.0	140.2

21 quant formats run end to end; the table shows a representative slice. Full report: `apps/octo/dist/octo-run-v0.76.txt`.

> **Honest low-bit note**
> High-bit formats (Q8, Q6, Q5, Q4, IQ4) stay coherent. Sub-2-bit formats (IQ1/IQ2, Ternary, Binary) are fast and tiny but degrade into gibberish. The bench keeps them in to show the speed, size, and quality trade honestly rather than hiding it.

Sample replies

```
Q8_0  / gpu   I'm a helpful assistant. I'm here to help you with your inquiries.
Q4_0  / gpu   I'm ready to help. What can I help you with?
Q6_K  / gpu   2 + 2 is 4.
Q6_K  / simd  I'm a helpful assistant.
IQ2_XXS/gpu   ,))hidAPAAPAAPAAPAAPA...   (sub-2-bit degrades)
```

Go example

examples/44-octo/main.go

Run: `cd welvet/examples/44-octo && source ../env.sh && go run .`

```go
package main

import "fmt"

func main() {
        fmt.Println("Octo is its own module — the engine never imports it.")
        fmt.Println("  cd apps/octo && go run .")
        fmt.Println("  download -> convert -> .entity -> chat / serve / bench")
}
```

Output

exit 0 · last run via `go run .`

```
Octo is its own module — the engine never imports it.
  cd apps/octo && go run .
  download -> convert -> .entity -> chat / serve / bench
```

Validate (harness)

```
cd apps/octo && go run .
```

CHAPTER 68

Cameral sandwiches + AAI Lucy

github.com/openfluke/welvet/layers/parallel ✅ cameral

Why it exists

A single Dense chain cannot host two independent weight copies that share an input, merge, and optionally train under different updates. That is the cameral graph: hemispheres, not screen-space sprites and not a second hidden size.

What it is

Hemispheres / Bicameral / Sandwich / Mix. Stem → Parallel mid → Dense head. SetBranchModes + TrainStackMSE. Lucy races in AAI (test41 / test48 / test50) import this API; measuring is lucy/; harness is not engine.

```
x  →  Dense stem  →  ┬→ Hemi 1 (own W, own TrainMode)
                     ├→ Hemi 2
                     └→ Hemi n     merge add|avg|concat|filter
                              → Dense head → ŷ
                              MSE → g_y → each branch's update
```

Sandwich: stem and head are shared; hemispheres are sibling copies.

Why cameral exists

Welvet already has Sequential (ordered compose) and Residual (skip). Neither is "two brains on one x." A hemisphere is an independent sub-net with its own weights. Bi / Tri / Quad are n hemispheres plus a merge. **Mix** stamps a different `TrainMode` per hemi (`SetBranchModes`) so left can be StepBP while right is TweenChain — one loss on the merge. Grid `training.Step` applies one mode to the whole net; Mix needs `TrainStackMSE` or the BranchModes stamp is a lie.

The sandwich (stem → mid → head) is not extra depth for its own sake. Credit modes need a *head Jacobian* (or a W_head^T proxy). Without a head, FastProxy / HeadProxy have nothing to inject. Stem gets the down-going vector after hemispheres merge. That is why AAI Lucy jobs are sandwiches even when cameral count is 1 (mid is a single Dense, not Parallel).

API (engine, public)

- `Hemispheres` / `HemispheresFrom` — n twins; merge add/avg/concat/filter
- `Bicameral` / `BicameralFrom` / `Sandwich` — stem → Parallel → head Stack
- `SetBranchModes` / `TrainStackMSE` — Mix path
- `ResidualGraft` — y = F(x)+x when F is Parallel
- `WriteCameralFile` / `LoadCameral` — cameral `.entity` (in `model/entity`)
- `CamSync` — optional weight averaging across cams / layers → §70

Layer package chapter: §27. Named updates: §67. Weight sync across cams (and across mesh layouts when shapes match): §70.

AAI Lucy benches

AAI is a separate tree. It `replace`s `github.com/openfluke/welvet` (public chaosglue module). The engine does not import AAI. Lucy *math* is §66.

Bench	What it answers
test41	Native cameral Lucy (Bi/Tri/Mix) on toys — does the sandwich train at all.
test48	Credit sweep: layers × dtypes × short jobs. Home of the equations in §67.
test50	Deep FP32 race: all 29 named modes × Dense/Bi/Tri × $1^3/2^3/3^3$ origin-only. Rival = hard Acc vs StepBP.

test50 copy: Split / Alt / MeshSplit family about **+5 to +12 Acc** over StepBP (~68–74%). Sine: Acc ceiling (many modes 100% including StepBP); FastProxy SoftAcc ~56–68 vs StepBP ~42–53. XOR: almost everyone at 75% (3 of 4 bits) — parking lot. Sparse wins Score, not Acc. Origin-only cubes: extra cells are disabled; 3^3 is hop topology, not 27 copies.

w2a Test49 is the *permutation smoke* of those 29 modes (in `[0] Run ALL`), not a Lucy race. Do not quote Test49 as “we beat backprop.”

Go example

examples/68-cameral/main.go

Run: `cd welvet/examples/68-cameral && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/parallel"
        "github.com/openfluke/welvet/quant"
)

func main() {
        s, err := parallel.Bicameral(8, 16, 1, core.ActivationLeakyReLU,
                core.DTypeFloat32, quant.FormatNone)
        if err != nil {
                panic(err)
        }
        hemi := s.Children[1].(*parallel.Layer)
        hemi.SetBranchModes(parallel.ModeStepBP, parallel.ModeTweenSplitFastProxy)

        x := core.NewTensor[float32](1, 8)
        t := core.NewTensor[float32](1, 1)
        t.Data[0] = 0.5
        loss, err := parallel.TrainStackMSE(s, x, t, parallel.ModeStepBP, 0.01)
        fmt.Println("mix loss", loss, "err", err)
}
```

Output

exit 0 · last run via `go run .`

```
mix loss 0 err <nil>
```

CHAPTER 70

CamSync — inter-cameral / cross-mesh weight blend

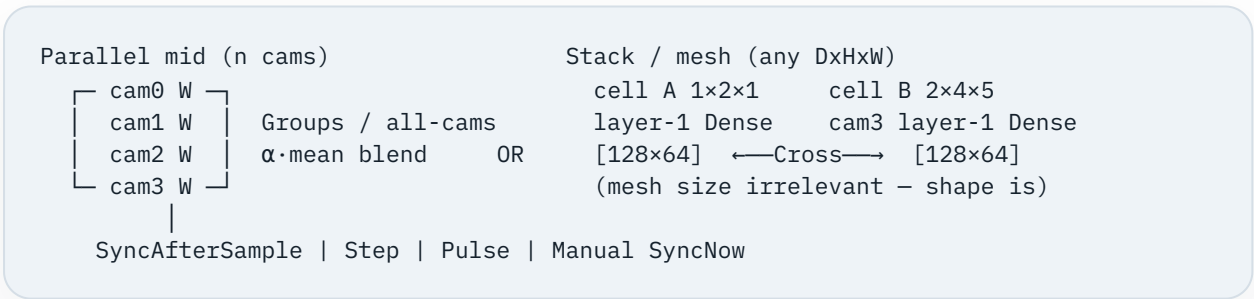
github.com/openfluke/welvet/layers/parallel ✅ CamSync

Why it exists

Mix BranchModes and different inits let cams diverge on purpose. Sometimes you want them to share — gently (1% pull) or hard (full average) — within a Parallel, across Stack children, or even between same-shaped stores sitting on wildly different mesh layouts (1×2×1 ↔ 2×4×5). That is CamSync: couple weights without collapsing the graph into one Dense.

What it is

CamSyncConfig{Alpha, When, Groups, Cross}. BlendStores pulls every store in a clique toward the mean. Empty Groups = all cams; Cross wires SyncEndpoint pairs across layers/cams. Shape rule: Rows×Cols must match. Bidirectional today; one-way teacher→student not yet.



Same-shaped stores sync. Mesh topology is host wiring via Groups + Cross.

The insane bit

CamSync does not care that one Parallel lives on a `1×2×1` cell and another store sits on cam 3 of a `2×4×5`. Name both endpoints; if `Rows×Cols` match, they blend. Volumetric layout is configuration — not a built-in grid type inside CamSync.

Knobs

Field	Meaning
`Enabled` / `Alpha`	Soft pull `w ← (1−α)·w + α·mean`. `0.01` = 1%, `1.0` = hard sync. Default α is 1 when Enabled and Alpha ≤ 0.
`When`	`SyncManual` · `SyncAfterSample` · `SyncAfterStep` · `SyncAfterPulse`. Hooks fire from `TrainStackMSE` / `CE` and `Pulse()`.
`Groups`	Within one Parallel: cam-index cliques. Empty ⇒ one group of every branch. Example: `{{0,1},{2,3}}` syncs pairs only.
`Cross`	`[]SyncPair` of `SyncEndpoint{StackIdx, Branch, Store}` — same layer across cams, Dense stem ↔ matching cam Dense, or any same-shaped pair the Stack can resolve.

What works / what does not (yet)

- ✅ All cams together, selective groups, soft or hard α
- ✅ Cross-layer / cross-cam pairs inside one Stack
- ✅ “Same layer on a different mesh” when weight shapes match
- ✅ **v1.1.1** — cam adjusting on **all layer types** (Dense, CNN, Parallel, Stack-resolved stores)
- ❌ One-directional (teacher → student) — pairs are mutual mean-blend
- ❌ Mismatched `Rows×Cols` — rejected or skipped
- ❌ Auto sync across *two separate Stack instances* — host calls `weights.BlendStores` on the two stores

API

- `parallel.CamSyncConfig` · `DefaultCamSync()` · `SetCamSync` on Layer or Stack
- `SyncNow` / `Pulse` / `MaybeSync`
- `weights.BlendStores` · `weights.StoreCosine` (diagnostics)

Cameral sandwiches: §68. Parallel package: §27. Host probe: `ok/cam_sync` (MNIST sample α×cams matrix).

Go example

examples/70-cam-sync/main.go

Run: `cd welvet/examples/70-cam-sync && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/dense"
	"github.com/openfluke/welvet/layers/parallel"
	"github.com/openfluke/welvet/quant"
	"github.com/openfluke/welvet/stub/seed"
	"github.com/openfluke/welvet/weights"
)

func main() {
	s, err := parallel.Bicameral(8, 16, 1, core.ActivationLeakyReLU,
		core.DTypeFloat32, quant.FormatNone)
	if err != nil {
		panic(err)
	}
	hemi := s.Children[1].(*parallel.Layer)
	left := hemi.Branches[0].(*dense.Layer).Weights
	right := hemi.Branches[1].(*dense.Layer).Weights
	_ = seed.InitStoreHe(left, 16, seed.From("cam0"))
	_ = seed.InitStoreHe(right, 16, seed.From("cam1"))

	hemi.SetCamSync(parallel.CamSyncConfig{
		Enabled: true,
		Alpha:   1.0,
		When:    parallel.SyncManual,
	})
	before, _ := weights.StoreCosine(left, right)
	if err := hemi.SyncNow(); err != nil {
		panic(err)
	}
	after, _ := weights.StoreCosine(left, right)
	fmt.Printf("cosine before=%.4f after=%.4f\n", before, after)
}
```

Output

exit 0 · last run via `go run .`

```
cosine before=-0.0697 after=1.0000
```

CHAPTER 45

stub/seed

github.com/openfluke/welvet/stub/seed 

Why it exists

Ship topology recipes (layer seeds → He-init) without weight blobs.

What it is

SeedFrom/From, RNG, InitStoreHe, Dense/MHA/SwiGLU/CNN infinite manifests.

Go example

examples/45-stub-seed/main.go

Run: `cd welvet/examples/45-stub-seed && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/stub/seed"
)

func main() {
        s := seed.From("demo-net", 0)
        rng := seed.New(s)
        fmt.Println(rng.NormFloat64())
}
```

Output

exit 0 · last run via `go run .`

```
0.4118750632366808
```

CHAPTER 46

stub/serialization

github.com/openfluke/welvet/stub/serialization 

Why it exists

Volumetric grids need JSON/ENTITY persist beyond transformer packs.

What it is

SerializeEntity/LoadEntity, SerializeGrid/GridFromSpec — native FormatNone bytes; packed via wire.

Go example

examples/46-stub-serialization/main.go

Run: `cd welvet/examples/46-stub-serialization && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/architecture"
	"github.com/openfluke/welvet/stub/serialization"
)

func main() {
	g := architecture.NewGrid(1, 1, 1, 1)
	raw, err := serialization.SerializeGrid(g)
	fmt.Println(len(raw), err)
}
```

Output

exit 0 · last run via `go run .`

```
101 <nil>
```

CHAPTER 47

stub/memory

github.com/openfluke/welvet/stub/memory 

Why it exists

HF→ENTITY and GPU upload need footprint accounting and optional history charts.

What it is

FromGrid, Footprint, InitScavenger, ReleaseTransient; WELVET_MEMORY_HISTORY=1.

Go example

examples/47-stub-memory/main.go

Run: `cd welvet/examples/47-stub-memory && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/architecture"
        "github.com/openfluke/welvet/stub/memory"
)

func main() {
        g := architecture.NewGrid(1, 1, 1, 1)
        fp := memory.FromGrid(g)
        fmt.Printf("%+v\n", fp)
}
```

Output

exit 0 · last run via `go run .`

```
{HostWeightsMB:0 GPUWeightsMB:0 GPUKVMB:0}
```

CHAPTER 48

stub/donate

github.com/openfluke/welvet/stub/donate 

Why it exists

LAN donors should accept framed JSON jobs without embedding HTTP in the engine.

What it is

u32-LE + JSON frames; ServeTCP/Dial; model_push vs local_lm. v0 workers echo.

Go example

examples/48-stub-donate/main.go

Run: `cd welvet/examples/48-stub-donate && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/stub/donate"
)

func main() {
        fmt.Println("default port", donate.DefaultPort)
        // ln, err := donate.ServeTCP(donate.ServerOptions{Addr: ":17001", Mode:
donate.ServerLocalLM})
}
```

Output

exit 0 · last run via `go run .`

```
default port 17001
```

CHAPTER 49

stub/fountain

github.com/openfluke/welvet/stub/fountain 

Why it exists

Recover specialist weight blobs over lossy links via LT fountain codes, then ensemble.

What it is

NewLTEncoder/Decoder, RecoverWeightBlobs, Neural/DenseFactory helpers.

Go example

examples/49-stub-fountain/main.go

Run: `cd welvet/examples/49-stub-fountain && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/stub/fountain"
)

func main() {
        src := [][]byte{make([]byte, 64), make([]byte, 64), make([]byte, 64)}
        enc, err := fountain.NewLTEncoder(src, 42)
        fmt.Printf("%T %v\n", enc, err)
}
```

Output

exit 0 · last run via `go run .`

```
*fountain.LTEncoder <nil>
```

CHAPTER 50

stub/hardware

github.com/openfluke/welvet/stub/hardware 

Why it exists

Dispatchers and UIs need a portable host audit (OS/CPU/RAM/GPU).

What it is

Audit() → SystemAudit with linux /proc or platform fallbacks.

Go example

examples/50-stub-hardware/main.go

Run: `cd welvet/examples/50-stub-hardware && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/stub/hardware"
)

func main() {
        a := hardware.Audit()
        fmt.Printf("%+v\n", a.CPU)
        fmt.Println(hardware.Description())
}
```

Output

exit 0 · last run via `go run .`

```
{Model:Apple M5 Logical:10 GOMAXPROCS:10}
OS: darwin | CPU: Apple M5 (10) | RAM: 0.00 GB | GPU: Unknown GPU (Apple
```

```
  Metal/Universal) (0.0 GB)
```

CHAPTER 51

stub/accel — NPU/Metal/QNN

github.com/openfluke/welvet/stub/accel 

Why it exists

Vendor accelerators plug beside WebGPU — not replace it. Qualcomm HTP is live via `nativeqnn` + sibling `welvet.qnn` (see book §79–81).

What it is

stub/accel remains a reserved plugin slot ( ). Real Hexagon path: `github.com/openfluke/welvet/nativeqnn` · BackendQNN · SyncToQNN.

Go example

examples/51-stub-accel/main.go

Run: `cd welvet/examples/51-stub-accel && source ../env.sh && go run .`

```
package main

import "fmt"

// stub/accel has no symbols yet — HTP lives in nativeqnn (book §79–81).
func main() {
        fmt.Println("stub/accel: reserved; use nativeqnn for Qualcomm HTP")
}
```

Output

exit 0 · last run via `go run .`

```
stub/accel: reserved; use nativeqnn for Qualcomm HTP
```

CHAPTER 52

stub/clustering

github.com/openfluke/welvet/stub/clustering 

Why it exists

Offline clustering helpers on tensors without inventing a second math stack.

What it is

KMeansCluster, HierarchicalGroup, distance/silhouette helpers.

Go example

examples/52-stub-clustering/main.go

Run: `cd welvet/examples/52-stub-clustering && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/stub/clustering"
)

func main() {
        pts := []*core.Tensor[float32]{
                core.NewTensor[float32](2),
                core.NewTensor[float32](2),
        }
        pts[0].Data[0], pts[0].Data[1] = 0, 0
        pts[1].Data[0], pts[1].Data[1] = 1, 1
        cent, assign := clustering.KMeansCluster(pts, 2, 10, false)
        fmt.Println(len(cent), assign)
}
```

Output

exit 0 · last run via `go run .`

```
2 [1 0]
```

CHAPTER 53

stub/ensemble

github.com/openfluke/welvet/stub/ensemble 

Why it exists

Combine multiple model votes and find complementary specialists.

What it is

MajorityVote, FindComplementaryMatches, PerformanceSimilarity.

Go example

examples/53-stub-ensemble/main.go

Run: `cd welvet/examples/53-stub-ensemble && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/stub/ensemble"
)

func main() {
        votes := [][]int{{0, 1, 1}, {0, 0, 1}, {0, 1, 0}}
        fmt.Println(ensemble.MajorityVote(votes))
}
```

Output

exit 0 · last run via `go run .`

```
[0 1 1]
```

CHAPTER 54

stub/evaluation

github.com/openfluke/welvet/stub/evaluation

Why it exists

Benchmark grids through runtime/forward with deviation metrics.

What it is

EvaluateNetwork, MultiNetworkEvaluation, DeviationMetrics.

Go example

examples/54-stub-evaluation/main.go

Run: `cd welvet/examples/54-stub-evaluation && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/architecture"
	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/stub/evaluation"
)

func main() {
	g := architecture.NewGrid(1, 1, 1, 1)
	in := []*core.Tensor[float32]{core.NewTensor[float32](1, 4)}
	rep, err := evaluation.EvaluateNetwork(g, in, []float64{0, 0, 0, 0})
	fmt.Println(rep, err)
}
```

Output

exit 0 · last run via `go run .`

```
<nil> evaluation: length mismatch inputs=1 expected=4
```

CHAPTER 55

stub/grafting

github.com/openfluke/welvet/stub/grafting 

Why it exists

Merge grids into Parallel/Residual structures for topology experiments.

What it is

GraftGrids, ResidualGraft, GraftToGrid — Dense branches in v0.

Go example

examples/55-stub-grafting/main.go

Run: `cd welvet/examples/55-stub-grafting && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/architecture"
        "github.com/openfluke/welvet/layers/parallel"
        "github.com/openfluke/welvet/stub/grafting"
)

func main() {
        a, b := architecture.NewGrid(1, 1, 1, 1), architecture.NewGrid(1, 1, 1, 1)
        out, err := grafting.GraftGrids([]*architecture.Grid{a, b},
parallel.CombineConcat)
        fmt.Println(out != nil, err)
}
```

Output

exit 0 · last run via `go run .`

```
false grafting: no valid branches
```

CHAPTER 56

stub/grouping

github.com/openfluke/welvet/stub/grouping 

Why it exists

Detect layer archetypes from safetensor-style names before mounting.

What it is

GroupRelatedTensors, DetectMHA/DetectSwiGLU/DetectRMSNorm → ArchetypeHint.

Go example

examples/56-stub-grouping/main.go

Run: `cd welvet/examples/56-stub-grouping && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/stub/grouping"
)

func main() {
	tensors := []grouping.DetectedTensor{
		{Name: "model.layers.0.self_attn.q_proj.weight"},
		{Name: "model.layers.0.self_attn.k_proj.weight"},
		{Name: "model.layers.0.self_attn.v_proj.weight"},
		{Name: "model.layers.0.self_attn.o_proj.weight"},
	}
	ok, hint := grouping.DetectMHA("block0", tensors, 64, 4)
	fmt.Println(ok, hint)
}
```

Output

exit 0 · last run via `go run .`

```
true {MultiHeadAttention 64 4}
```

CHAPTER 57

stub/introspection

github.com/openfluke/welvet/stub/introspection 

Why it exists

UIs and FFI need to list Grid methods without hardcoding every export.

What it is

GetMethods, GetMethodsJSON, GetMethodSignature.

Go example

examples/57-stub-introspection/main.go

Run: `cd welvet/examples/57-stub-introspection && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/architecture"
        "github.com/openfluke/welvet/stub/introspection"
)

func main() {
        g := architecture.NewGrid(1, 1, 1, 1)
        methods, err := introspection.GetMethods(g)
        fmt.Println(len(methods), err)
}
```

Output

exit 0 · last run via `go run .`

```
9 <nil>
```

CHAPTER 58

stub/observer

github.com/openfluke/welvet/stub/observer 

Why it exists

Attach forward/backward observers for debugging without coupling to tanhi UDP.

What it is

Observer interface; ConsoleObserver / HTTPObserver / BufferObserver; ComputeLayerStats.

Go example

examples/58-stub-observer/main.go

Run: `cd welvet/examples/58-stub-observer && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/stub/observer"
)

func main() {
        var obs observer.Observer = &observer.ConsoleObserver{}
        fmt.Printf("%T\n", obs)
}
```

Output

exit 0 · last run via `go run .`

```
*observer.ConsoleObserver
```

CHAPTER 59

stub/pipeline

github.com/openfluke/welvet/stub/pipeline 

Why it exists

Decoder wavefront stats helpers — not a full Lucy-style pipeline runner yet.

What it is

PipelineForwardStats, TokenTimelineSummary.

Go example

examples/59-stub-pipeline/main.go

Run: `cd welvet/examples/59-stub-pipeline && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/stub/pipeline"
)

func main() {
        var s pipeline.PipelineForwardStats
        fmt.Printf("%+v\n", s)
}
```

Output

exit 0 · last run via `go run .`

```
{PipelineTicks:0 SubLayerOps:0 MaxActiveJobs:0 MaxBlockSpread:0 MaxDistinctBlocks:0
MaxPendingTokens:0 StallFallback:false TokenDoneTick:[]}
```

CHAPTER 60

stub/templates

github.com/openfluke/welvet/stub/templates 

Why it exists

Chat prompts must match model families (ChatML, Llama3, BitNet) without app-specific string glue.

What it is

Template.BuildPrompt; presets ChatML, Llama3, BitNetInstruction, MicrosoftBitNetChat.

Go example

examples/60-stub-templates/main.go

Run: `cd welvet/examples/60-stub-templates && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/stub/templates"
)

func main() {
        p := templates.ChatML.BuildPrompt(nil, "You are helpful.", "Say hi")
        fmt.Println(p)
}
```

Output

exit 0 · last run via `go run .`

```
<|im_start|>system
You are helpful.
<|im_start|>user
Say hi
<|im_start|>assistant
```

CHAPTER 61

stub/universal

github.com/openfluke/welvet/stub/universal 

Why it exists

Probe unknown safetensor geometry and mount placeholder grids until full weight import lands.

What it is

LoadUniversal / LoadUniversalDetailed, ProbeDeepGeometry, MountGeometrically.

Go example

examples/61-stub-universal/main.go

Run: `cd welvet/examples/61-stub-universal && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/stub/universal"
)

func main() {
        g, err := universal.LoadUniversal("/path/to/snapshot")
        fmt.Println(g != nil, err)
}
```

Output

exit 0 · last run via `go run .`

```
false universal: "/path/to/snapshot": open /path/to/snapshot: no such file or
directory
```

CHAPTER 62

w2a — validation harness

github.com/openfluke/w2a ✅ harness

Why it exists

Engine packages must stay free of tests. w2a owns timed 34×20×3 matrices, gap census, honesty stamps, and the train-mode permutation smoke (Test49). See §63 for a live full-suite run.

What it is

Interactive go run . ([0] Run ALL). Suites under suites/*. StampBackendNote / AffinePackable prevent fake ✅ . Test49: AllNamedTrainModes × 1³/2³/3³ × Parallel/Bicameral/poly, origin-only — included in [0].

Go example

examples/62-w2a/main.go

Run: `cd welvet/examples/62-w2a && source ../env.sh && go run .`

```
package main

import "fmt"

func main() {
	fmt.Println("w2a is a separate module — engine packages never contain
tests.")
	fmt.Println("")
	fmt.Println("  cd w2a")
	fmt.Println("  go run .                                          # interactive;
[0] = ALL")
	fmt.Println("  go test ./tests/dense -v")
	fmt.Println("  go test ./tests/parallel -run Test49AllTrainModesCubes -
count=1 -v")
}
```

Output

exit 0 · last run via `go run .`

```
w2a is a separate module — engine packages never contain tests.

  cd w2a
  go run .                                        # interactive; [0] = ALL
  go test ./tests/dense -v
  go test ./tests/parallel -run Test49AllTrainModesCubes -count=1 -v
```

Validate (harness)

```
cd w2a && go run .
cd w2a && go test ./tests/dense -v && go test ./tests/parallel -run
Test49AllTrainModesCubes -count=1 -v
```

CHAPTER 63

Validation report — full suite

github.com/openfluke/w2a ✅ 246k cells

Why it exists

Claims are cheap; a stamped matrix is not. This is the actual output of one full w2a [0] Run ALL so the book's ✅ marks are backed by numbers you can reproduce, not asserted.

What it is

Every timed layer sweeps its dtype × format × backend matrix; every suite runs its case checks. v1.0 board: 246,032 matrix cells, FAIL 0, RESULT PASS. GAP cells are declared skips (not silent fails).

> **Full suite: PASS**
> `[0] Run ALL` —
> **246,032**
> matrix cells — FAIL 0 — RESULT: PASS. GAP remains (honesty stamps): GDN non-float32 / non-packable formats, AffinePacked, exotic hemi9 dtypes, census + cross-numeric train declared skips, a handful of ser cells. GAP ≠ fail. Case-only stubs (seed, serialization, memory, donate, fountain, hardware) stay off the engine scorecard.

What grew since the 228k board

- **Test49** — 29 named TrainModes × $1^3/2^3/3^3$ × Parallel + Bicameral + poly kinds (origin-only). All cells OK.
- **Cameral perm** — Mix / BranchModes / credit on sandwiches (OK cells + declared GAP on exotic dtypes).
- **Nested Sequential / Residual** mixed children + `ResidualGraft`.
- **Mesh*** credit tokens on the named-mode list (stack family on a grid walk).

Do not quote this stamp as a Lucy race or “beats PyTorch.” It is surface coverage: every claimed path either runs or is stamped GAP. Lucy Acc vs Score lives in AAI test50 (§68).

Cross-numeric train

W2A Step includes **Cross-Numeric Train** (kinds × weight dtype × act host; full census ~10.7k under the `train` op). Storage truth checked after `StepMesh` — no retained f32 master for non-float32 FormatNone. Public Dense demotion + W×A showcase: down-the-dem (chapter 65).

Go example

examples/63-validation/main.go

Run: `cd welvet/examples/63-validation && source ../env.sh && go run .`

```
package main

import "fmt"

func main() {
        cells := 246032
        fmt.Printf("w2a [0] ALL: %d cells  FAIL 0  RESULT PASS\n", cells)
        fmt.Println("GAP = declared skip (GDN non-f32, AffinePacked, …) — not a
fail")
}
```

Output

exit 0 · last run via `go run .`

```
w2a [0] ALL: 246032 cells  FAIL 0  RESULT PASS
GAP = declared skip (GDN non-f32, AffinePacked, …) — not a fail
```

Validate (harness)

```
cd w2a && go run .   # [0] Run ALL; writes logs/suite.txt
```

CHAPTER 64

Scorecard → v1.0 / minors

v1.7.0

Why it exists

Version is earned from a weighted board, not marketing. v1.0 is 100/100 on the engine board. Minor/patch tags (v1.1.0, v1.1.1, …) pack features without a new board. Apps, stubs, and NPU sit off-board.

What it is

version = 0.{round(earned)} until 100 → v1.0. Scorecard today **100/100**; this book tags **v1.7.0**. Training credit is §9 on the board (8 pts), not an afterthought.

§	Area	Wt	Earned
1	Foundation	15	15
2	Dense MatVec microkernel	15	15
3	Transformer stack	14	14
4	CNN / RNN / LSTM	6	6
5	Extended layers (GDN, ConvT, Mamba, Parallel, …)	7	7
6	Runtime + architecture	8	8
7	Systems	5	5
8	Model / IO	8	8
9	Training credit (29 named TrainModes + BranchModes)	8	8
10	Peak fused / no host ALU	14	14
	Total → v1.0 board	100	**100**

v1.0 cut

- **w2a [0] ALL** — 246,032 cells, FAIL 0, RESULT PASS (§63)
- **Training credit** — StepBP, Tween*, Split, Alt, HeadProxy, FastProxy, Linear, Sparse, Step* pipe twins, Mesh* (§67)
- **Cameral** — Hemispheres, Mix BranchModes, Sandwich, ResidualGraft, cameral .entity (§68)
- **CamSync** — soft/hard inter-cameral + cross-layer same-shape weight blend (§70)
- **Nested Sequential/Residual** mixed children (Dense/SwiGLU/RMSNorm/LayerNorm)
- **lucy** — SoftAcc / Availability / Score + `BuildLPD` density (§66 · §69)
- **v1.0.3** — Dense FormatNone SIMD *forward*: WireF64/`DotTileF64`, expand-once, BF16 convert; ~4.3× geo-mean on 34-dtype deep suite vs prior Go; Dense-backed projs inherit (§6 · §11)
- **v1.1.1** — Cam adjusting fixed: CamSync on **all layers** (§70)
- **v1.1.0** — CamSync + `training_modes.md` + Lucy Lean density + book §70 (§70 · §69)
- **v1.1.7** — Full NativeCPU TrainMode C ABI (stack/parallel/grid/line) + `SyncWeightsFromNative` (§76 · §77 · §78)
- **v1.7.0** — Apple MLX forward-only GPU via `nativemlx` + independent welvet.mlx **1.0.0**. Explicit `SyncToMLX` / `BackendMLX` / `PreferMLX` on **Apple silicon macOS**; no TrainMSE on MLX; caps reject unsupported mounts. §110–113; Go engine remains 100/100.
- **v1.6.0** — Vulkan GPU via `nativevulkan` + independent welvet.vulkan **0.79** (ABI 1). Explicit mount/train/SyncToGo on **Linux x86_64**; JSON Go/C++/Vulkan routing on Linux amd64. §103–109; Go engine remains 100/100. Pure-Vulkan contract: no silent host fallback labeled as Vulkan.
- **v1.5.0** — AMD GPU via `nativerocm` + independent welvet.rocm **0.79** (ABI 1). Explicit mount/train/SyncToGo on **Linux x86_64 + Windows amd64**; JSON Go/C++/ROCm routing on Linux amd64. §96–102; Go engine remains 100/100. **GPU compute untested** until a ROCm-capable HSA device is available.
- **v1.4.0** — NVIDIA GPU via `nativecuda` + independent welvet.cuda **0.79** (ABI 1). Explicit mount/train/SyncToGo on **Linux x86_64 + Windows amd64**; JSON Go/C++/CUDA routing on Linux amd64. §89–95; Go engine remains 100/100.
- **v1.3.0** — Apple GPU via `nativemetal` + independent welvet.metal (ABI 1). Explicit mount/train/SyncToGo and JSON Go/C++/Metal routing. §82–88.
- **v1.2.0** — Qualcomm Hexagon HTP via `nativeqnn` + welvet.qnn **1.0.0** (opt-in `BackendQNN`, layer SyncToQNN, large-shape matrix in w2a) (§79 · §80 · §81)

> **Off this board**
>
> `apps/octo`, `stub/*`, and vendor NPU/GPU scorecard credit (HTP ships as sibling `welvet.qnn` — see §79; CUDA as `welvet.cuda` — see §89; ROCm as `welvet.rocm` —

> see §96; Vulkan as `welvet.vulkan` — see §103; MLX as `welvet.mlx` — see §110 · sibling github.com/openfluke/welvet.mlx). FastProxy can match/beat StepBP *Acc* on sine/copy toys; Sparse wins Lucy *Score* via Avail. Do not write “Sparse is better backprop.” Engine v1 ≠ “beats PyTorch.”

Go example

examples/64-scorecard/main.go

Run: `cd welvet/examples/64-scorecard && source ../env.sh && go run .`

```
package main

import "fmt"

func main() {
        earned := 100.0
        fmt.Println("v1.7.0")
        fmt.Printf("scorecard %.0f/100\n", earned)
}
```

Output

exit 0 · last run via `go run .`

```
v1.7.0
scorecard 100/100
```

CHAPTER 76

NativeCPU — opt-in welvet.cpp C ABI

github.com/openfluke/welvet/nativecpp ✅ linux/amd64

Why it exists

Go stays the default engine. When you want the C++ MatVec / NormalBP / TrainMSE path in-process, set `BackendNativeCPU` (never Prefer as default). Pending C++ weights stay off Go Stores until `SyncWeightsFromNative` — required before `.entity` write.

What it is

linux/amd64 + cgo ships `native/cpp/linux_amd64/libwelvet.so` (engine 0.1.85). `NativeCPUAvailable()`, `PreferNativeCPU()`, JSON `ExecConfig`, Dense Forward/Backward + `TrainNativeMSE` + Sync.

> **Default stays BackendCPUTiled**
>
> `core.DefaultBackend()` is Go tiled. `PreferNativeCPU()` returns NativeCPU only when the shared lib loads — still opt-in on each layer via `Exec.Backend` or `ApplyExecConfig`.

Related: every signed layer train → §77. TrainMode dispatchers (stack / parallel / mesh / line) → §78. Runnable copies also live in openfluke/example `welvet/76-nativecpu`.

Go example

examples/76-nativecpu/main.go

Run: `cd welvet/examples/76-nativecpu && source ../env.sh && go run .`

```
package main

import (
        "fmt"
```

```
        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/dense"
        "github.com/openfluke/welvet/nativecpp"
        "github.com/openfluke/welvet/quant"
)

func main() {
        fmt.Println("version", nativecpp.Version())
        fmt.Println("available", nativecpp.NativeCPUAvailable())
        fmt.Println("default", core.DefaultBackend())
        fmt.Println("prefer", core.PreferNativeCPU())
        if !nativecpp.NativeCPUAvailable() {
                fmt.Println("SKIP: need linux/amd64 libwelvet")
                return
        }

        init := make([]float32, 4*4)
        for i := range init {
                init[i] = float32(i+1) * 0.05
        }
        l, err := dense.NewConfigured(4, 4, core.ActivationLinear, core.DTypeFloat32,
quant.FormatNone, init)
        must(err)
        l.Exec.Backend = core.BackendNativeCPU

        cfg, err := nativecpp.ParseExecConfigJSON([]byte(`{"default":"native_cpu"}`))
        must(err)
        l.ApplyExecConfig(cfg, "main", "0,0,0,0", "0")
        fmt.Println("exec", l.Exec.Backend)

        x := core.NewTensor[float32](2, 4)
        for i := range x.Data {
                x.Data[i] = float32(i+1) * 0.1
        }
        pre, post, err := dense.Forward(l, x)
        must(err)
        _, gW, err := dense.Backward(l, post, x, pre)
        must(err)
        fmt.Println("fwd/bwd", len(post.Data), len(gW.Data))

        tgt := core.NewTensor[float32](2, 4)
        for i := range tgt.Data {
                tgt.Data[i] = 0.25
        }
        loss, err := dense.TrainNativeMSE(l, x, tgt, 0.05)
        must(err)
        fmt.Println("pending", l.NativeWeightsPending(), "loss", loss)
        must(l.SyncWeightsFromNative())
        fmt.Println("synced", !l.NativeWeightsPending())
}

func must(err error) {
        if err != nil {
                panic(err)
        }
}
```

Output

exit 0 · last run via `go run .`

```
version 0.1.85-m2-grid-line-trainmodes
available true
default cpu-tiled
prefer native_cpu
exec native_cpu
fwd/bwd 8 16
pending true loss 0.6012000681936766
synced true
```

Validate (harness)

```
cd welvet/examples/76-nativecpu && source ../env.sh && go run .
```

CHAPTER 77

NativeCPU — TrainNativeMSE per layer

github.com/openfluke/welvet/layers ✅ signed kinds

Why it exists

Each signed layer has a packed C ABI NormalBP path. One smoke walks Dense → CNN/ConvT → norms → RNN/LSTM → Sequential/Residual → Parallel → Embedding → Softmax → KMeans → MHA/SwiGLU → Mamba/GDN/Metacognition with Capture + Sync.

What it is

Set `Exec.Backend = BackendNativeCPU`, call package `TrainNativeMSE`, then `SyncWeightsFromNative` when Go Stores must match C++ (entity export).

Opt-in / Sync semantics: §76. Mode-aware stack/parallel/mesh/line: §78. Example tree: `example/welvet/77-native-layers`.

Go example

examples/77-native-layers/main.go

Run: `cd welvet/examples/77-native-layers && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/cnn1"
	"github.com/openfluke/welvet/layers/cnn2"
	"github.com/openfluke/welvet/layers/cnn3"
	"github.com/openfluke/welvet/layers/convt1"
	"github.com/openfluke/welvet/layers/convt2"
	"github.com/openfluke/welvet/layers/convt3"
	"github.com/openfluke/welvet/layers/dense"
	"github.com/openfluke/welvet/layers/embedding"
	"github.com/openfluke/welvet/layers/gdn"
	"github.com/openfluke/welvet/layers/kmeans"
```

```
        "github.com/openfluke/welvet/layers/layernorm"
        "github.com/openfluke/welvet/layers/lstm"
        "github.com/openfluke/welvet/layers/mamba"
        "github.com/openfluke/welvet/layers/metacognition"
        "github.com/openfluke/welvet/layers/mha"
        "github.com/openfluke/welvet/layers/parallel"
        "github.com/openfluke/welvet/layers/residual"
        "github.com/openfluke/welvet/layers/rmsnorm"
        "github.com/openfluke/welvet/layers/rnn"
        "github.com/openfluke/welvet/layers/sequential"
        "github.com/openfluke/welvet/layers/softmax"
        "github.com/openfluke/welvet/layers/swiglu"
        "github.com/openfluke/welvet/nativecpp"
        "github.com/openfluke/welvet/quant"
)

func main() {
        if !nativecpp.NativeCPUAvailable() {
                fmt.Println("SKIP: need linux/amd64 libwelvet")
                return
        }
        demos := []struct {
                name string
                fn   func() error
        }{
                {"dense", demoDense},
                {"rmsnorm", demoRMS},
                {"layernorm", demoLN},
                {"swiglu", demoSwiGLU},
                {"mha", demoMHA},
                {"softmax", demoSoftmax},
                {"embedding", demoEmbed},
                {"cnn1", demoCNN1},
                {"cnn2", demoCNN2},
                {"cnn3", demoCNN3},
                {"convt1", demoConvT1},
                {"convt2", demoConvT2},
                {"convt3", demoConvT3},
                {"rnn", demoRNN},
                {"lstm", demoLSTM},
                {"sequential", demoSeq},
                {"residual", demoRes},
                {"parallel", demoPara},
                {"kmeans", demoKMeans},
                {"mamba", demoMamba},
                {"gdn", demoGDN},
                {"metacognition", demoMeta},
        }
        ok := 0
        for _, d := range demos {
                if err := d.fn(); err != nil {
                        fmt.Println("FAIL", d.name, err)
                        continue
                }
                fmt.Println("OK  ", d.name)
                ok++
        }
        fmt.Printf("native-layers ok=%d/%d\n", ok, len(demos))
        if ok != len(demos) {
                panic("native-layers: some demos failed")
```

```
	}
}

func fill(shape ...int) *core.Tensor[float32] {
	t := core.NewTensor[float32](shape...)
	for i := range t.Data {
		t.Data[i] = 0.1 + float32(i%7)*0.03
	}
	return t
}

func tgtLike(x *core.Tensor[float32]) *core.Tensor[float32] {
	t := core.NewTensor[float32](x.Shape...)
	for i := range t.Data {
		t.Data[i] = 0.15
	}
	return t
}

func demoDense() error {
	l, err := dense.New(4, 4, core.ActivationLinear, core.DTypeFloat32)
	if err != nil {
		return err
	}
	l.Exec.Backend = core.BackendNativeCPU
	x := fill(2, 4)
	_, err = dense.TrainNativeMSE(l, x, tgtLike(x), 0.05)
	if err != nil {
		return err
	}
	return l.SyncWeightsFromNative()
}

func demoRMS() error {
	l, err := rmsnorm.New(rmsnorm.Config{Dim: 4})
	if err != nil {
		return err
	}
	l.Exec.Backend = core.BackendNativeCPU
	x := fill(2, 4)
	_, err = rmsnorm.TrainNativeMSE(l, x, tgtLike(x), 0.05)
	if err != nil {
		return err
	}
	return l.SyncWeightsFromNative()
}

func demoLN() error {
	l, err := layernorm.New(layernorm.Config{Dim: 4})
	if err != nil {
		return err
	}
	l.Exec.Backend = core.BackendNativeCPU
	x := fill(2, 4)
	_, err = layernorm.TrainNativeMSE(l, x, tgtLike(x), 0.05)
	if err != nil {
		return err
	}
	return l.SyncWeightsFromNative()
}
```

```
func demoSwiGLU() error {
	l, err := swiglu.New(swiglu.Config{InputDim: 4, IntermediateDim: 8})
	if err != nil {
		return err
	}
	l.Exec.Backend = core.BackendNativeCPU
	x := fill(1, 4)
	_, err = swiglu.TrainNativeMSE(l, x, tgtLike(x), 0.05)
	if err != nil {
		return err
	}
	return l.SyncWeightsFromNative()
}

func demoMHA() error {
	l, err := mha.New(mha.Config{DModel: 8, NumHeads: 2, NumKVHeads: 2, HeadDim:
4, MaxSeqLen: 8})
	if err != nil {
		return err
	}
	l.Exec.Backend = core.BackendNativeCPU
	x := fill(1, 2, 8)
	_, err = mha.TrainNativeMSE(l, x, tgtLike(x), 0.02)
	if err != nil {
		return err
	}
	return l.SyncWeightsFromNative()
}

func demoSoftmax() error {
	l, err := softmax.New(softmax.Config{Dim: 4})
	if err != nil {
		return err
	}
	l.Exec.Backend = core.BackendNativeCPU
	x := fill(2, 4)
	_, err = softmax.TrainNativeMSE(l, x, tgtLike(x), 0.05)
	return err
}

func demoEmbed() error {
	l, err := embedding.New(embedding.Config{VocabSize: 8, EmbeddingDim: 4,
SeqLen: 2})
	if err != nil {
		return err
	}
	l.Exec.Backend = core.BackendNativeCPU
	x := fill(1, 2)
	x.Data[0], x.Data[1] = 1, 2
	t := fill(1, 2, 4)
	_, err = embedding.TrainNativeMSE(l, x, t, 0.05)
	if err != nil {
		return err
	}
	return l.SyncWeightsFromNative()
}

func demoCNN1() error {
	l, err := cnn1.New(cnn1.Config{InChannels: 1, Filters: 2, SeqLen: 8, Kernel:
```

```
3})
        if err != nil {
                return err
        }
        l.Exec.Backend = core.BackendNativeCPU
        x := fill(1, 1, 8)
        _, post, err := cnn1.Forward(l, x)
        if err != nil {
                return err
        }
        _, err = cnn1.TrainNativeMSE(l, x, tgtLike(post), 0.02)
        if err != nil {
                return err
        }
        return l.SyncWeightsFromNative()
}

func demoCNN2() error {
        l, err := cnn2.New(cnn2.Config{InChannels: 1, Filters: 2, Height: 4, Width:
4, Kernel: 3})
        if err != nil {
                return err
        }
        l.Exec.Backend = core.BackendNativeCPU
        x := fill(1, 1, 4, 4)
        _, post, err := cnn2.Forward(l, x)
        if err != nil {
                return err
        }
        _, err = cnn2.TrainNativeMSE(l, x, tgtLike(post), 0.02)
        if err != nil {
                return err
        }
        return l.SyncWeightsFromNative()
}

func demoCNN3() error {
        l, err := cnn3.New(cnn3.Config{InChannels: 1, Filters: 2, Depth: 2, Height:
2, Width: 2, Kernel: 2})
        if err != nil {
                return err
        }
        l.Exec.Backend = core.BackendNativeCPU
        x := fill(1, 1, 2, 2, 2)
        _, post, err := cnn3.Forward(l, x)
        if err != nil {
                return err
        }
        _, err = cnn3.TrainNativeMSE(l, x, tgtLike(post), 0.02)
        if err != nil {
                return err
        }
        return l.SyncWeightsFromNative()
}

func demoConvT1() error {
        l, err := convt1.New(convt1.Config{InChannels: 1, Filters: 2, SeqLen: 4,
Kernel: 3})
        if err != nil {
                return err
```

```
        }
        l.Exec.Backend = core.BackendNativeCPU
        x := fill(1, 1, 4)
        _, post, err := convt1.Forward(l, x)
        if err != nil {
                return err
        }
        _, err = convt1.TrainNativeMSE(l, x, tgtLike(post), 0.02)
        if err != nil {
                return err
        }
        return l.SyncWeightsFromNative()
}

func demoConvT2() error {
        l, err := convt2.New(convt2.Config{InChannels: 1, Filters: 2, Height: 2,
Width: 2, Kernel: 2})
        if err != nil {
                return err
        }
        l.Exec.Backend = core.BackendNativeCPU
        x := fill(1, 1, 2, 2)
        _, post, err := convt2.Forward(l, x)
        if err != nil {
                return err
        }
        _, err = convt2.TrainNativeMSE(l, x, tgtLike(post), 0.02)
        if err != nil {
                return err
        }
        return l.SyncWeightsFromNative()
}

func demoConvT3() error {
        l, err := convt3.New(convt3.Config{InChannels: 1, Filters: 2, Depth: 2,
Height: 2, Width: 2, Kernel: 2})
        if err != nil {
                return err
        }
        l.Exec.Backend = core.BackendNativeCPU
        x := fill(1, 1, 2, 2, 2)
        _, post, err := convt3.Forward(l, x)
        if err != nil {
                return err
        }
        _, err = convt3.TrainNativeMSE(l, x, tgtLike(post), 0.02)
        if err != nil {
                return err
        }
        return l.SyncWeightsFromNative()
}

func demoRNN() error {
        l, err := rnn.New(rnn.Config{InputSize: 4, HiddenSize: 4, SeqLen: 2})
        if err != nil {
                return err
        }
        l.Exec.Backend = core.BackendNativeCPU
        x := fill(1, 2, 4)
        _, err = rnn.TrainNativeMSE(l, x, tgtLike(x), 0.05)
```

```
        if err != nil {
                return err
        }
        return l.SyncWeightsFromNative()
}

func demoLSTM() error {
        l, err := lstm.New(lstm.Config{InputSize: 4, HiddenSize: 4, SeqLen: 2})
        if err != nil {
                return err
        }
        l.Exec.Backend = core.BackendNativeCPU
        x := fill(1, 2, 4)
        _, err = lstm.TrainNativeMSE(l, x, tgtLike(x), 0.05)
        if err != nil {
                return err
        }
        return l.SyncWeightsFromNative()
}

func demoSeq() error {
        l, err := sequential.New(sequential.Config{Dim: 4, Depth: 2})
        if err != nil {
                return err
        }
        l.Exec.Backend = core.BackendNativeCPU
        x := fill(2, 4)
        _, err = sequential.TrainNativeMSE(l, x, tgtLike(x), 0.05)
        if err != nil {
                return err
        }
        return l.SyncWeightsFromNative()
}

func demoRes() error {
        l, err := residual.New(residual.Config{Dim: 4, Depth: 1})
        if err != nil {
                return err
        }
        l.Exec.Backend = core.BackendNativeCPU
        x := fill(2, 4)
        _, err = residual.TrainNativeMSE(l, x, tgtLike(x), 0.05)
        if err != nil {
                return err
        }
        return l.SyncWeightsFromNative()
}

func demoPara() error {
        init := make([]float32, 2*4*4)
        for i := range init {
                init[i] = 0.03 * float32(i+1)
        }
        l, err := parallel.NewConfigured(parallel.Config{Dim: 4, OutFeat: 4,
Branches: 2, Combine: parallel.CombineAdd},
                core.DTypeFloat32, quant.FormatNone, init, nil)
        if err != nil {
                return err
        }
        l.Exec.Backend = core.BackendNativeCPU
```

```
	l.SyncBranchExec()
	x := fill(2, 4)
	_, err = parallel.TrainNativeMSE(l, x, tgtLike(x), 0.05)
	if err != nil {
		return err
	}
	return l.SyncWeightsFromNative()
}

func demoKMeans() error {
	l, err := kmeans.New(kmeans.Config{NumClusters: 3, FeatureDim: 4})
	if err != nil {
		return err
	}
	l.Exec.Backend = core.BackendNativeCPU
	x := fill(2, 4)
	_, post, err := kmeans.Forward(l, x)
	if err != nil {
		return err
	}
	_, err = kmeans.TrainNativeMSE(l, x, tgtLike(post), 0.05)
	if err != nil {
		return err
	}
	return l.SyncWeightsFromNative()
}

func demoMamba() error {
	l, err := mamba.New(mamba.Config{DModel: 4, DState: 2, SeqLen: 2})
	if err != nil {
		return err
	}
	l.Exec.Backend = core.BackendNativeCPU
	x := fill(1, 2, 4)
	_, err = mamba.TrainNativeMSE(l, x, tgtLike(x), 0.05)
	if err != nil {
		return err
	}
	return l.SyncWeightsFromNative()
}

func demoGDN() error {
	l, err := gdn.New(gdn.Config{HiddenSize: 4, NumKeyHeads: 1, NumValueHeads: 1,
KeyHeadDim: 2, ValueHeadDim: 2, ConvKernel: 2})
	if err != nil {
		return err
	}
	l.Exec.Backend = core.BackendNativeCPU
	x := fill(1, 2, 4)
	_, err = gdn.TrainNativeMSE(l, x, tgtLike(x), 0.05)
	if err != nil {
		return err
	}
	return l.SyncWeightsFromNative()
}

func demoMeta() error {
	l, err := metacognition.New(metacognition.Config{Dim: 4})
	if err != nil {
		return err
```

```
        }
        l.Exec.Backend = core.BackendNativeCPU
        x := fill(2, 4)
        _, err = metacognition.TrainNativeMSE(l, x, tgtLike(x), 0.05)
        if err != nil {
                return err
        }
        return l.SyncWeightsFromNative()
}
```

Output

exit 0 · last run via `go run .`

```
OK   dense
OK   rmsnorm
OK   layernorm
OK   swiglu
OK   mha
OK   softmax
OK   embedding
OK   cnn1
OK   cnn2
OK   cnn3
OK   convt1
OK   convt2
OK   convt3
OK   rnn
OK   lstm
OK   sequential
OK   residual
OK   parallel
OK   kmeans
OK   mamba
OK   gdn
OK   metacognition
native-layers ok=22/22
```

Validate (harness)

```
cd welvet/examples/77-native-layers && source ../env.sh && go run .
```

CHAPTER 78

NativeCPU — TrainMode stack / parallel / mesh / line

github.com/openfluke/welvet/layers/parallel ✅ TrainMode C ABI

Why it exists

Leaf credit for every TrainMode family goes through one C dispatcher. Mesh clock and Line pipe have their own host-shape ABIs. Go intercepts `TrainStackMSE` / `TrainMSE` when `BackendNativeCPU`; call Sync before entity.

What it is

`welvet_stack_train_mse` · `welvet_parallel_train_mse` · `welvet_grid_train_mse` · `welvet_stack_line_train_mse` (engine 0.1.85). Families: BP, Tween, TweenChain, TweenSplit, TweenAlt, Freeze. Parallel BranchModes keep Adversarial / Memory gates inside C++.

Opt-in + Sync: §76. Per-layer NormalBP: §77. Named mode equations: §67. Example: `example/welvet/78-native-trainmodes`.

Go example

examples/78-native-trainmodes/main.go

Run: `cd welvet/examples/78-native-trainmodes && source ../env.sh && go run .`

```
package main

import (
        "fmt"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/dense"
        "github.com/openfluke/welvet/layers/parallel"
        "github.com/openfluke/welvet/nativecpp"
        "github.com/openfluke/welvet/quant"
)
```

```
func main() {
	if !nativecpp.NativeCPUAvailable() {
		fmt.Println("SKIP: need linux/amd64 libwelvet")
		return
	}
	demoStackFamilies()
	demoParallelModes()
	demoGridMesh()
	demoLineStep()
	fmt.Println("native-trainmodes done")
}

func demoStackFamilies() {
	modes := []parallel.TrainMode{
		parallel.ModeNormalBP, parallel.ModeTween, parallel.ModeTweenChain,
		parallel.ModeTweenSplit, parallel.ModeTweenAlt, parallel.ModeFreeze,
	}
	for _, mode := range modes {
		children := make([]any, 2)
		for i := 0; i < 2; i++ {
			init := make([]float32, 16)
			for j := range init {
				init[j] = 0.04 * float32(i*16+j+1)
			}
			d, err := dense.NewConfigured(4, 4, core.ActivationLinear,
core.DTypeFloat32, quant.FormatNone, init)
			must(err)
			children[i] = d
		}
		s, err := parallel.NewStack(children...)
		must(err)
		s.SetBackend(core.BackendNativeCPU)
		x, t := core.NewTensor[float32](2, 4), core.NewTensor[float32](2, 4)
		for i := range x.Data {
			x.Data[i] = 0.08 * float32(i+1)
			t.Data[i] = 0.3
		}
		loss, err := parallel.TrainStackMSE(s, x, t, mode, 0.05)
		must(err)
		must(s.SyncWeightsFromNative())
		fmt.Println("stack", mode, "loss", loss)
	}
}

func demoParallelModes() {
	for _, mode := range []parallel.TrainMode{parallel.ModeTween,
parallel.ModeAdversarial} {
		init := make([]float32, 2*4*4)
		for i := range init {
			init[i] = 0.02 * float32(i+1)
		}
		l, err := parallel.NewConfigured(parallel.Config{Dim: 4, OutFeat: 4,
Branches: 2, Combine: parallel.CombineAdd},
			core.DTypeFloat32, quant.FormatNone, init, nil)
		must(err)
		l.Exec.Backend = core.BackendNativeCPU
		l.SyncBranchExec()
		x, t := core.NewTensor[float32](2, 4), core.NewTensor[float32](2, 4)
		for i := range x.Data {
```

```
                        x.Data[i] = 0.07 * float32(i+1)
                        t.Data[i] = 0.25
                }
                loss, err := parallel.TrainMSE(l, x, t, mode, 0.08)
                must(err)
                must(l.SyncWeightsFromNative())
                fmt.Println("parallel", mode, "loss", loss)
        }
}

func demoGridMesh() {
        const dim, batch, ticks = 3, 2, 2
        packed := make([]float32, 2*dim*dim)
        for i := range packed {
                packed[i] = 0.05 * float32(i+1)
        }
        wAfter := make([]float32, len(packed))
        x := make([]float32, batch*dim)
        tgt := make([]float32, batch*dim)
        for i := range x {
                x[i] = 0.1 * float32(i+1)
                tgt[i] = 0.3
        }
        loss, err := nativecpp.GridTrainMSE(
                int32(parallel.ModeMeshBP), 1, 1, 2, 1, batch, ticks, dim,
                1, 0, int32(core.ActivationLinear), int32(core.DTypeFloat32), 0,
                packed, x, tgt, 0.1, wAfter,
        )
        must(err)
        fmt.Println("grid MeshBP loss", loss)
}

func demoLineStep() {
        const nChildren, nTicks, batch, dim = 2, 3, 2, 3
        inH := []int32{dim, dim}
        outH := []int32{dim, dim}
        act := []int32{int32(core.ActivationLinear), int32(core.ActivationLinear)}
        packed := make([]float32, nChildren*dim*dim)
        for i := range packed {
                packed[i] = 0.04 * float32(i+1)
        }
        wAfter := make([]float32, len(packed))
        xTicks := make([]float32, nTicks*batch*dim)
        tgtTicks := make([]float32, nTicks*batch*dim)
        for i := range xTicks {
                xTicks[i] = 0.05 * float32(i+1)
                tgtTicks[i] = 0.2
        }
        loss, err := nativecpp.StackLineTrainMSE(
                int32(parallel.ModeStepTween), nChildren, nTicks, batch, 0.1,
int32(core.DTypeFloat32),
                inH, outH, act, packed, xTicks, tgtTicks, wAfter,
        )
        must(err)
        fmt.Println("line StepTween loss", loss)
}

func must(err error) {
        if err != nil {
                panic(err)
```

```
        }
}
```

Output

exit 0 · last run via `go run .`

```
stack NormalBP loss 3.9041736651040533
stack Tween loss 3.9041736651040533
stack TweenChain loss 3.9041736651040533
stack TweenSplit loss 3.9041736651040533
stack TweenAlt loss 3.5210903517464196
stack Freeze loss 3.9041736651040533
parallel Tween loss 0.5521819071927838
parallel Adversarial loss 0.5521819071927838
grid MeshBP loss 0.3168250438054419
line StepTween loss 0.10267065279439665
native-trainmodes done
```

Validate (harness)

```
cd welvet/examples/78-native-trainmodes && source ../env.sh && go run .
```

CHAPTER 79

QNN — opt-in Hexagon HTP (welvet.qnn)

github.com/openfluke/welvet/nativeqnn ✅ windows/arm64 + cgo

Why it exists

Qualcomm Hexagon HTP is a sibling engine (`welvet.qnn` 1.0.0), not a silent fallback inside Go. Opt-in `BackendQNN` + explicit `SyncToQNN` keep CPU/SIMD/WebGPU as defaults. Caps reject unsupported layer/dtype mounts.

What it is

`nativeqnn.Available()`, `Version()`, `SupportedLayers()`, Dense `SyncToQNN` → `PrepareQNN` → `Forward` with `BackendQNN`. FormatNone host weights; HTP may run f16/i8/... inside the DLL.

Needs **Windows ARM64**, cgo, and shipped `libwelvet_qnn.dll` (plus Qualcomm QAIRT runtime on PATH). Other OS/arch → `Available()==false` and examples print SKIP.

Layer tour: §80. Large-shape matrix + % wins: §81. NativeCPU twin: §76.

```
Go host (FormatNone f32)     SyncToQNN / mirror      welvet.qnn DLL
  Dense / Softmax / …  ──►  caps-check + Capture  ──►  HTP graphExecute
  BackendQNN Forward   ◄──  unpack y to Tensor    ◄──  dtype inside NPU
```

Opt-in path — never Prefer as default.

Go example

examples/79-qnn/main.go

Run: `cd welvet/examples/79-qnn && source ../env.sh && go run .`

```
package main

import (
        "fmt"
```

```
        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/dense"
        "github.com/openfluke/welvet/nativeqnn"
        "github.com/openfluke/welvet/quant"
)

func main() {
        fmt.Println("version", nativeqnn.Version())
        fmt.Println("available", nativeqnn.Available())
        fmt.Println("default", core.DefaultBackend())
        if !nativeqnn.Available() {
                fmt.Println("SKIP: need windows/arm64 + libwelvet_qnn (+ QAIRT)")
                return
        }
        fmt.Println("layers", nativeqnn.SupportedLayers())

        init := make([]float32, 4*4)
        for i := range init {
                init[i] = float32(i+1) * 0.05
        }
        l, err := dense.NewConfigured(4, 4, core.ActivationLinear, core.DTypeFloat32,
quant.FormatNone, init)
        must(err)
        must(l.SyncToQNN())
        must(l.PrepareQNN(2))
        l.Exec.Backend = core.BackendQNN

        x := core.NewTensor[float32](2, 4)
        for i := range x.Data {
                x.Data[i] = float32(i+1) * 0.1
        }
        _, post, err := dense.Forward(l, x)
        must(err)
        fmt.Println("fwd", len(post.Data), "backend", l.Exec.Backend)
        fmt.Println("qnn done")
}

func must(err error) {
        if err != nil {
                panic(err)
        }
}
```

Output

exit 0 · last run via `go run .`

```
version
available false
default cpu-tiled
SKIP: need windows/arm64 + libwelvet_qnn (+ QAIRT)
```

Validate (harness)

```
cd welvet/examples/79-qnn && source ../env.sh && go run .
```

CHAPTER 80

QNN — HTP layer tour (session ABI)

github.com/openfluke/welvet/nativeqnn

✅ 12 layer forwards

Why it exists

w2a `go_v_qnn` exercises every shipped HTP forward. Book examples use the same session ABI with tiny shapes so a Snapdragon box can smoke-test without the full matrix.

What it is

Open HTP session once, then Dense (prepare+exec), Softmax, RMSNorm, LayerNorm, CNN2/3, ConvT2/3, Embedding, Residual, SwiGLU, MHA. LSTM is listed in caps but large shapes FAIL execute today — see §81. Layer packages with SyncToQNN: dense, softmax, rmsnorm, layernorm, sequential.

Opt-in / Sync: §79. Timing honesty (DLL exec vs Go→NPU call): §81. Full matrix lives in `w2a/suites/go_v_qnn`.

Go example

examples/80-qnn-layers/main.go

Run: `cd welvet/examples/80-qnn-layers && source ../env.sh && go run .`

```
package main

import (
	"fmt"

	"github.com/openfluke/welvet/nativeqnn"
)

func main() {
	if !nativeqnn.Available() {
		fmt.Println("SKIP: need windows/arm64 + libwelvet_qnn (+ QAIRT)")
		return
	}
```

```
        sess, err := nativeqnn.DefaultSession()
        must(err)

        demoDense(sess)
        demoSoftmax(sess)
        demoRms(sess)
        demoLN(sess)
        demoCnn2(sess)
        demoCnn3(sess)
        demoConvT2(sess)
        demoConvT3(sess)
        demoEmbed(sess)
        demoResidual(sess)
        demoSwiGlu(sess)
        demoMha(sess)
        fmt.Println("qnn-layers done")
}

func demoDense(s *nativeqnn.Session) {
        const batch, in, out = 2, 4, 4
        W := fill(in * out)
        x, y := fill(batch*in), make([]float32, batch*out)
        must(s.DensePrepareF32(batch, in, out, W, nil))
        must(s.DenseExecuteF32(x, y))
        fmt.Println("dense", len(y))
}

func demoSoftmax(s *nativeqnn.Session) {
        const rows, cols = 2, 8
        x, y := fill(rows*cols), make([]float32, rows*cols)
        must(s.SoftmaxForwardF32(rows, cols, x, y))
        fmt.Println("softmax", len(y))
}

func demoRms(s *nativeqnn.Session) {
        const b, seq, dim = 1, 2, 4
        g, x, y := ones(dim), fill(b*seq*dim), make([]float32, b*seq*dim)
        must(s.RmsNormForwardF32(b, seq, dim, 1e-5, g, x, y))
        fmt.Println("rmsnorm", len(y))
}

func demoLN(s *nativeqnn.Session) {
        const b, seq, dim = 1, 2, 4
        g, beta := ones(dim), make([]float32, dim)
        x, y := fill(b*seq*dim), make([]float32, b*seq*dim)
        must(s.LayerNormForwardF32(b, seq, dim, 1e-5, g, beta, x, y))
        fmt.Println("layernorm", len(y))
}

func demoCnn2(s *nativeqnn.Session) {
        // HTP path today is 1×1 (W = cout*cin), same as w2a go_v_qnn.
        const n, h, w, cin, cout = 1, 4, 4, 2, 2
        W, bias := fill(cout*cin), make([]float32, cout)
        x, y := fill(n*h*w*cin), make([]float32, n*h*w*cout)
        must(s.Cnn2ForwardF32(n, h, w, cin, cout, W, bias, x, y))
        fmt.Println("cnn2", len(y))
}

func demoCnn3(s *nativeqnn.Session) {
        const n, d, h, w, cin, cout = 1, 2, 2, 2, 2, 2
```

```
        W, bias := fill(cout*cin), make([]float32, cout)
        x, y := fill(n*d*h*w*cin), make([]float32, n*d*h*w*cout)
        must(s.Cnn3ForwardF32(n, d, h, w, cin, cout, W, bias, x, y))
        fmt.Println("cnn3", len(y))
}

func demoConvT2(s *nativeqnn.Session) {
        const n, h, w, cin, cout = 1, 4, 4, 2, 2
        W, bias := fill(cout*cin), make([]float32, cout)
        x, y := fill(n*h*w*cin), make([]float32, n*h*w*cout)
        must(s.ConvT2ForwardF32(n, h, w, cin, cout, W, bias, x, y))
        fmt.Println("convt2", len(y))
}

func demoConvT3(s *nativeqnn.Session) {
        const n, d, h, w, cin, cout = 1, 2, 2, 2, 2, 2
        W, bias := fill(cout*cin), make([]float32, cout)
        x, y := fill(n*d*h*w*cin), make([]float32, n*d*h*w*cout)
        must(s.ConvT3ForwardF32(n, d, h, w, cin, cout, W, bias, x, y))
        fmt.Println("convt3", len(y))
}

func demoEmbed(s *nativeqnn.Session) {
        const vocab, dim, n = 16, 4, 4
        table := fill(vocab * dim)
        ids := []int32{0, 1, 2, 3}
        y := make([]float32, n*dim)
        must(s.EmbeddingForwardF32(vocab, dim, n, table, ids, y))
        fmt.Println("embedding", len(y))
}

func demoResidual(s *nativeqnn.Session) {
        const n = 16
        a, b, y := fill(n), fill(n), make([]float32, n)
        must(s.ResidualForwardF32(n, a, b, y))
        fmt.Println("residual", len(y))
}

func demoSwiGlu(s *nativeqnn.Session) {
        const batch, in, hidden = 1, 4, 8
        wG, wU, wD := fill(hidden*in), fill(hidden*in), fill(in*hidden)
        x, y := fill(batch*in), make([]float32, batch*in)
        must(s.SwiGluForwardF32(batch, in, hidden, wG, wU, wD, x, y))
        fmt.Println("swiglu", len(y))
}

func demoMha(s *nativeqnn.Session) {
        const batch, seq, dim, heads = 1, 2, 8, 2
        w := fill(dim * dim)
        x, y := fill(batch*seq*dim), make([]float32, batch*seq*dim)
        must(s.MhaForwardF32(batch, seq, dim, heads, w, w, w, w, x, y))
        fmt.Println("mha", len(y))
}

func fill(n int) []float32 {
        out := make([]float32, n)
        for i := range out {
                out[i] = 0.01 * float32(i+1)
        }
        return out
```

```
}

func ones(n int) []float32 {
        out := make([]float32, n)
        for i := range out {
                out[i] = 1
        }
        return out
}

func must(err error) {
        if err != nil {
                panic(err)
        }
}
```

Output

exit 0 · last run via `go run .`

```
SKIP: need windows/arm64 + libwelvet_qnn (+ QAIRT)
```

Validate (harness)

```
cd welvet/examples/80-qnn-layers && source ../env.sh && go run .
```

CHAPTER 81

QNN — go_v_qnn RESULTS (large shapes)

github.com/openfluke/w2a/suites/go_v_qnn ✅ matrix published

Why it exists

Toy shapes lie. The w2a large matrix (dense 32×512×512, softmax 64×4096, …) is the honest Snapdragon X story: NPU DLL `graphExecute` vs Go/native f32 CPU, plus Go→NPU call overhead (cgo + pack + online compose).

What it is

Published tables in `w2a/suites/go_v_qnn/RESULTS.md`. Dense prepare-once wins ~93–98% vs Go/native. Softmax/norms/CNN/SwiGLU/MHA win on DLL exec; online layers pay compose cost on the call timer. LSTM large FAIL. Host compares f32 only — dtype matrix is NPU-side.

How to read the timers

- **NPU DLL exec** — mono inside welvet.qnn around `graphExecute` only.
- **Go→NPU call** — full C ABI from Go (cgo + pack/unpack + online create/finalize).
- **Dense** = prepare-once + warmed execute; other layers = online compose today.
- **%** = `(slower-faster)/slower×100` vs that host column.

Headline NPU wins (DLL exec vs Go f32)

Layer	dtype	Go	NPU	vs Go
dense	u8	3.05ms	164µs	NPU **95%**
dense	f32	3.05ms	201µs	NPU **93%**
softmax	f16	4.31ms	446µs	NPU **90%**
rmsnorm	u8	9.40ms	897µs	NPU **90%**
layernorm	f16	6.82ms	553µs	NPU **92%**

Layer	dtype	Go	NPU	vs Go
cnn2	i8	12.56ms	763µs	NPU **94%**
swiglu	u16	1.92ms	352µs	NPU **82%**
mha	f16	4.51ms	372µs	NPU **92%**

Full status / call overhead / NPU-wins-only tables: RESULTS.md. Re-run: `cd w2a && go test ./tests/go_v_qnn -v -count=1 -run TestGoVQNNMatrix -timeout 180m`.

Failures today: **lstm** float32/float16 — `code=2 execute` on large shapes.

Go example

examples/81-qnn-results/main.go

Run: `cd welvet/examples/81-qnn-results && source ../env.sh && go run .`

```
package main

import "fmt"

// Snapshot of w2a/suites/go_v_qnn/RESULTS.md (2026-10-23, Snapdragon X).
// This example is host-only — no DLL required.
func main() {
        fmt.Println("welvet.qnn 1.0.0 · go_v_qnn large matrix")
        fmt.Println("host: Windows ARM64 · shapes: dense 32x512x512 …")
        fmt.Println()
        fmt.Println("NPU DLL wins (subset vs Go f32):")
        rows := []struct{ layer, dt, goT, npu, pct string }{
                {"dense", "u8", "3.05ms", "164µs", "95%"},
                {"dense", "f32", "3.05ms", "201µs", "93%"},
                {"softmax", "f16", "4.31ms", "446µs", "90%"},
                {"rmsnorm", "u8", "9.40ms", "897µs", "90%"},
                {"layernorm", "f16", "6.82ms", "553µs", "92%"},
                {"cnn2", "i8", "12.56ms", "763µs", "94%"},
                {"swiglu", "u16", "1.92ms", "352µs", "82%"},
                {"mha", "f16", "4.51ms", "372µs", "92%"},
        }
        for _, r := range rows {
                fmt.Printf("  %-10s %-4s  Go %s  NPU %s  → %s\n", r.layer, r.dt,
r.goT, r.npu, r.pct)
        }
        fmt.Println()
        fmt.Println("note: call timer » DLL exec on online layers (compose each
call)")
        fmt.Println("fail: lstm f32/f16 execute code=2")
```

```
        fmt.Println("full: w2a/suites/go_v_qnn/RESULTS.md")
}
```

Output

exit 0 · last run via `go run .`

```
welvet.qnn 1.0.0 · go_v_qnn large matrix
host: Windows ARM64 · shapes: dense 32x512x512 …

NPU DLL wins (subset vs Go f32):
  dense      u8    Go 3.05ms  NPU 164µs  → 95%
  dense      f32   Go 3.05ms  NPU 201µs  → 93%
  softmax    f16   Go 4.31ms  NPU 446µs  → 90%
  rmsnorm    u8    Go 9.40ms  NPU 897µs  → 90%
  layernorm  f16   Go 6.82ms  NPU 553µs  → 92%
  cnn2       i8    Go 12.56ms  NPU 763µs  → 94%
  swiglu     u16   Go 1.92ms  NPU 352µs  → 82%
  mha        f16   Go 4.51ms  NPU 372µs  → 92%

note: call timer » DLL exec on online layers (compose each call)
fail: lstm f32/f16 execute code=2
full: w2a/suites/go_v_qnn/RESULTS.md
```

Validate (harness)

```
cd welvet/examples/81-qnn-results && source ../env.sh && go run .
```

CHAPTER 82

Apple Metal — mount and forward

github.com/openfluke/welvet/nativemetal `verified Apple M5`

Why it exists

Run a Go layer on Apple GPU with an explicit lifetime.

What it is

Mount an identity Dense layer, inspect the Metal version and device, and forward a Float32 tensor.

Session ownership

Create a session with `nativemetal.Mount`, call `Forward` repeatedly and close it when finished. Mount accepts supported layers, Stack, Parallel and Grid layouts. The library is independent of welvet.cpp.

Requirements and backend boundaries

Go Welvet **v1.3.0** uses the independent **welvet.metal 0.68** library (mounted model ABI 1, segment extension 1). Install the library from its checkout:

```
cd ../welvet.metal
./scripts/macos_install_go.sh
```

Execution requires Apple silicon macOS and Go with cgo. Set `WELVET_METAL_LIBRARY` to override library discovery. Ordinary Go CPU use on Linux or Windows does not require Apple hardware. Unsupported hosts print SKIP in these examples; missing libraries on supported hosts report an error.

Go schedules work and manages memory; numerical Metal forward, backward and weight updates execute custom Apple GPU kernels. This is GPU acceleration through Metal, not Apple Neural Engine execution. Activations are Float32; FormatNone

parameter storage supports 34 dtypes, with GDN restricted to Float32. Small examples demonstrate APIs; GPU launch costs can outweigh useful work at small shapes.

`cpu-tiled` means Go CPU; `native_cpu` means the separate welvet.cpp library; `metal` means the separate Apple GPU library. Neither native engine depends on the other. Examples 82–86 can use `go run -tags welvet_metal .` to exclude C++ linkage. Chapter 87 requires the normal cgo build for both native engines.

Go example

examples/82-apple-metal/main.go

Run: `cd welvet/examples/82-apple-metal && source ../env.sh && go run .`

```
// Mount one layer, execute its forward pass on Apple Metal, and release it.
package main

import (
	"flag"
	"fmt"
	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/dense"
	"github.com/openfluke/welvet/nativemetal"
	"github.com/openfluke/welvet/quant"
	"log"
	"runtime"
)

func main() {
	if e := run(); e != nil {
		log.Fatal(e)
	}
}
func run() error {
	library := flag.String("library", "", "optional libwelvet_metal.dylib path")
	flag.Parse()
	if runtime.GOOS != "darwin" || runtime.GOARCH != "arm64" {
		fmt.Println("SKIP: Apple Metal needs Apple silicon macOS and cgo")
		return nil
	}
	l, e := dense.NewConfigured(2, 2, core.ActivationLinear, core.DTypeFloat32,
quant.FormatNone, []float32{1, 0, 0, 1})
	if e != nil {
		return e
	}
	session, e := nativemetal.Mount(l, nativemetal.Options{LibraryPath:
*library})
	if e != nil {
		return e
	}
	defer session.Close()
```

```
        x := &core.Tensor[float32]{Shape: []int{1, 2}, Data: []float32{.5, -.25}}
        post, e := session.Forward(x)
        if e != nil {
                return e
        }
        fmt.Printf("backend=metal version=%s\ndevice=%s\ninput=%v output=%v
shape=%v\n", session.Version(), session.Identity(), x.Data, post.Data, post.Shape)
        return nil
}
```

Output

exit 0 · last run via `go run .`

```
backend=metal version=0.71
device=metal:Apple M5:reg=4294968546:ws=19069665280
input=[0.5 -0.25] output=[0.5 -0.25] shape=[1 2]
```

Validate (harness)

```
cd welvet/examples/82-apple-metal && source ../env.sh && go run .
```

CHAPTER 83

Apple Metal — Mamba parameter dtypes

github.com/openfluke/welvet/nativemetal `verified Apple M5`

Why it exists

Exercise native storage formats without changing the activation interface.

What it is

Run a seeded Mamba forward pass across all 34 native parameter dtypes, or select one with -dtype.

Storage versus arithmetic

`-dtype all` sweeps native parameter representations. It does not claim native Float64 GPU arithmetic or integer activation tensors: the Metal execution interface uses Float32 activations. Quantized FormatNone dtype storage and packed quantization formats are different coverage claims.

```
go run . -dtype all
go run . -dtype float16
```

Requirements and library installation: chapter 82.

Go example

examples/83-apple-mamba-dtypes/main.go

Run: `cd welvet/examples/83-apple-mamba-dtypes && source ../env.sh && go run .`

```
// Float32 tensors with independently selected native parameter encodings.
package main

import (
```

```
        "flag"
        "fmt"
        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/mamba"
        "github.com/openfluke/welvet/nativemetal"
        "github.com/openfluke/welvet/quant"
        "log"
        "runtime"
        "strings"
)

func main() {
        if e := run(); e != nil {
                log.Fatal(e)
        }
}
func run() error {
        selected := flag.String("dtype", "all", "native parameter dtype name, or all
34")
        flag.Parse()
        if runtime.GOOS != "darwin" || runtime.GOARCH != "arm64" {
                fmt.Println("SKIP: Apple Metal needs Apple silicon macOS and cgo")
                return nil
        }
        count := 0
        for _, dt := range core.AllDTypes {
                if *selected != "all" && !strings.EqualFold(*selected, dt.String()) {
                        continue
                }
                if e := forward(dt); e != nil {
                        return fmt.Errorf("%s: %w", dt, e)
                }
                count++
        }
        if count == 0 {
                return fmt.Errorf("unknown dtype %q", *selected)
        }
        fmt.Printf("Mamba dtypes=%d; activations=float32; parameters=FormatNone
native dtype\n", count)
        return nil
}
func forward(dt core.DType) error {
        const d, inner = 8, 16
        in, out := make([]float32, 2*inner*d), make([]float32, d*inner)
        for i := range in {
                in[i] = float32(i%17-8) * .01
        }
        for i := range out {
                out[i] = float32(i%13-6) * .01
        }
        model, e := mamba.NewConfigured(mamba.Config{DModel: d, DState: 4, Expand: 2,
SeqLen: 3}, dt, quant.FormatNone, in, out, nil, nil)
        if e != nil {
                return e
        }
        s, e := nativemetal.Mount(model)
        if e != nil {
                return e
        }
        defer s.Close()
```

```
        x := core.NewTensor[float32](1, 3, d)
        for i := range x.Data {
                x.Data[i] = float32(i%7-3) * .05
        }
        post, e := s.Forward(x)
        if e != nil {
                return e
        }
        var checksum float64
        for _, v := range post.Data {
                checksum += float64(v)
        }
        fmt.Printf("%-10s shape=%v checksum=%.8g\n", dt, post.Shape, checksum)
        return nil
}
```

Output

exit 0 · last run via `go run .`

```
float64    shape=[1 3 8] checksum=-0.016678239
float32    shape=[1 3 8] checksum=-0.016678239
float16    shape=[1 3 8] checksum=-0.016669162
bfloat16   shape=[1 3 8] checksum=-0.016553075
fp8e4m3    shape=[1 3 8] checksum=-0.017004712
fp8e5m2    shape=[1 3 8] checksum=-0.016270325
int64      shape=[1 3 8] checksum=-0.016678241
int32      shape=[1 3 8] checksum=-0.016678239
int16      shape=[1 3 8] checksum=-0.016677922
int8       shape=[1 3 8] checksum=-0.016622871
uint64     shape=[1 3 8] checksum=-0.016678238
uint32     shape=[1 3 8] checksum=-0.01667824
uint16     shape=[1 3 8] checksum=-0.01667843
uint8      shape=[1 3 8] checksum=-0.016727349
int4       shape=[1 3 8] checksum=-0.01510581
uint4      shape=[1 3 8] checksum=-0.017548652
fp4        shape=[1 3 8] checksum=0
int2       shape=[1 3 8] checksum=-0.032299955
uint2      shape=[1 3 8] checksum=-0.015402747
ternary    shape=[1 3 8] checksum=-0.02989793
binary     shape=[1 3 8] checksum=-0.0070436111
int        shape=[1 3 8] checksum=-0.016678241
uint       shape=[1 3 8] checksum=-0.016678238
uintptr    shape=[1 3 8] checksum=-0.016678238
complex64  shape=[1 3 8] checksum=-0.016678239
complex128 shape=[1 3 8] checksum=-0.016678239
nf4        shape=[1 3 8] checksum=-0.016108489
fp6        shape=[1 3 8] checksum=-0.01633804
int6       shape=[1 3 8] checksum=-0.01633804
uint6      shape=[1 3 8] checksum=-0.016878628
int5       shape=[1 3 8] checksum=-0.016140702
uint5      shape=[1 3 8] checksum=-0.017025853
int3       shape=[1 3 8] checksum=-0.018383351
uint3      shape=[1 3 8] checksum=-0.017474993
Mamba dtypes=34; activations=float32; parameters=FormatNone native dtype
```

Validate (harness)

```
cd welvet/examples/83-apple-mamba-dtypes && source ../env.sh && go run .
```

CHAPTER 84

Apple Metal — training modes

github.com/openfluke/welvet/nativemetal `verified Apple M5`

Why it exists

Keep repeated training updates in the mounted Metal session.

What it is

Train two Dense layers using one of the 29 numbered modes, with persistent Step Line state and real Grid layouts for Mesh modes.

Training families

`-mode all` exercises all 29 modes. Step modes retain their Line session across calls. Mesh modes use a three-cell Grid with disabled ingress and two active layers, advancing four ticks per call. These small demonstrations validate usage; they are not performance measurements.

```
go run . -mode NormalBP -steps 8
go run . -mode all -steps 8
```

Explicit mounted sessions keep updated weights native until `SyncToGo()`. Close discards any updates not returned to Go. See chapter 85 before saving.

Requirements and library installation: chapter 82.

Go example

examples/84-apple-training/main.go

Run: `cd welvet/examples/84-apple-training && source ../env.sh && go run .`

```
// Native Metal MSE training: persistent Stack Line state or an actual Grid clock.
package main

import (
        "flag"
        "fmt"
        "github.com/openfluke/welvet/architecture"
        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/dense"
        "github.com/openfluke/welvet/layers/parallel"
        "github.com/openfluke/welvet/nativemetal"
        "github.com/openfluke/welvet/quant"
        "log"
        "runtime"
)

func main() {
        if e := run(); e != nil {
                log.Fatal(e)
        }
}
func run() error {
        selected := flag.String("mode", "NormalBP", "training mode name, or all 29")
        steps := flag.Int("steps", 8, "training calls; retain the session for Step
pipeline state")
        flag.Parse()
        if *steps < 1 {
                return fmt.Errorf("steps must be positive")
        }
        if runtime.GOOS != "darwin" || runtime.GOARCH != "arm64" {
                fmt.Println("SKIP: Apple Metal needs Apple silicon macOS and cgo")
                return nil
        }
        if *selected == "all" {
                for mode := parallel.ModeNormalBP; mode <=
parallel.ModeStepTweenSplitSparse; mode++ {
                        if e := train(mode, *steps); e != nil {
                                return e
                        }
                }
                return nil
        }
        mode, e := parallel.ParseTrainMode(*selected)
        if e != nil {
                return e
        }
        if mode < parallel.ModeNormalBP || mode > parallel.ModeStepTweenSplitSparse {
                return fmt.Errorf("choose one of the 29 update modes")
        }
        return train(mode, *steps)
}
func train(mode parallel.TrainMode, steps int) error {
        var ops []any
        for layer := 0; layer < 2; layer++ {
                w := make([]float32, 16)
                for i := range w {
                        w[i] = float32(i%5-2) * .04
                }
                l, e := dense.NewConfigured(4, 4, core.ActivationTanh,
```

```
core.DTypeFloat32, quant.FormatNone, w)
                if e != nil {
                        return e
                }
                ops = append(ops, l)
        }
        stack, e := parallel.NewStack(ops...)
        if e != nil {
                return e
        }
        var model any = stack
        schedule := "Stack"
        if mode.RequiresGrid() {
                g := architecture.NewGrid(3, 1, 1, 1)
                g.Cells[0].Layer.IsDisabled = true
                for i, op := range ops {
                        g.Cells[i+1].Layer = op.(*dense.Layer).Core
                        g.Cells[i+1].Op = op
                }
                model = g
                schedule = "Grid mesh clock (4 ticks/call)"
        } else if mode.IsLineStep() {
                schedule = "Stack Line pipeline"
        }
        s, e := nativemetal.Mount(model)
        if e != nil {
                return e
        }
        defer s.Close()
        x := core.NewTensor[float32](2, 4)
        for i := range x.Data {
                x.Data[i] = float32(i%5-2) * .1
        }
        target := core.NewTensor[float32](2, 4)
        var loss float64
        for i := 0; i < steps; i++ {
                loss, e = s.TrainMSE(x, target, mode, .01, 4)
                if e != nil {
                        return fmt.Errorf("%s: %w", mode, e)
                }
        }
        if e = s.SyncToGo(); e != nil {
                return e
        }
        fmt.Printf("mode=%s schedule=%s calls=%d loss=%.8g weights=synced-to-Go\n",
mode, schedule, steps, loss)
        return nil
}
```

Output

exit 0 · last run via `go run .`

```
mode=NormalBP schedule=Stack calls=8 loss=6.7954462e-06 weights=synced-to-Go
```

Validate (harness)

```
cd welvet/examples/84-apple-training && source ../env.sh && go run .
```

CHAPTER 85

Apple Metal — weight return and ENTITY save

github.com/openfluke/welvet/nativemetal verified Apple M5

Why it exists

Return trained weights to Go before executing or saving the Go model.

What it is

Train natively, explicitly SyncToGo, save an ENTITY, reload it and compare the native parameter snapshots.

Weight handoff

```
session, err := nativemetal.Mount(layer)
// Check err; defer session.Close().
// Repeat session.TrainMSE(...) as needed.
err = session.SyncToGo()
// Check err, then save through Go ENTITY IO.
```

Mount copies model state into a native session. The original Go parameters stay unchanged while that explicit session trains. `SyncToGo()` returns parameter bytes, scales and Float64 biases for Go saving. This saves model parameters, not a portable checkpoint of live Line, proxy, KV or rule state.

Use `-out /path/model.entity` to choose the output location. The default uses a temporary directory and prints it.

Requirements and library installation: chapter 82.

Go example

examples/85-apple-save-weights/main.go

Run: `cd welvet/examples/85-apple-save-weights && source ../env.sh && go run .`

```
// Explicit mount ownership, weight return to Go, ENTITY save and reload.
package main

import (
	"flag"
	"fmt"
	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/dense"
	"github.com/openfluke/welvet/layers/parallel"
	"github.com/openfluke/welvet/model/entity"
	"github.com/openfluke/welvet/nativemetal"
	"github.com/openfluke/welvet/quant"
	"log"
	"os"
	"path/filepath"
	"reflect"
	"runtime"
)

func main() {
	if e := run(); e != nil {
		log.Fatal(e)
	}
}
func run() error {
	out := flag.String("out", "", "ENTITY destination; default is a new temporary
directory")
	flag.Parse()
	if runtime.GOOS != "darwin" || runtime.GOARCH != "arm64" {
		fmt.Println("SKIP: Apple Metal needs Apple silicon macOS and cgo")
		return nil
	}
	l, e := dense.NewConfigured(2, 2, core.ActivationLinear, core.DTypeFloat32,
quant.FormatNone, []float32{.3, -.2, .1, .4})
	if e != nil {
		return e
	}
	model, e := parallel.NewStack(l)
	if e != nil {
		return e
	}
	before, e := l.Weights.FlattenF32()
	if e != nil {
		return e
	}
	s, e := nativemetal.Mount(model)
	if e != nil {
		return e
	}
	defer s.Close()
	x := &core.Tensor[float32]{Shape: []int{1, 2}, Data: []float32{.5, .25}}
	target := core.NewTensor[float32](1, 2)
	for i := 0; i < 5; i++ {
		if _, e = s.TrainMSE(x, target, parallel.ModeNormalBP, .1); e != nil
{
			return e
```

```
                }
        }
        unchanged, e := l.Weights.FlattenF32()
        if e != nil {
                return e
        }
        if !reflect.DeepEqual(before, unchanged) {
                return fmt.Errorf("explicit mount unexpectedly changed Go parameters
before SyncToGo")
        }
        fmt.Println("Go parameters unchanged before SyncToGo")
        if e = s.SyncToGo(); e != nil {
                return e
        }
        after, e := l.Weights.FlattenF32()
        if e != nil {
                return e
        }
        if reflect.DeepEqual(before, after) {
                return fmt.Errorf("expected trained parameters after SyncToGo")
        }
        if *out == "" {
                dir, e := os.MkdirTemp("", "welvet-apple-entity-")
                if e != nil {
                        return e
                }
                *out = filepath.Join(dir, "trained.entity")
        }
        if e = entity.WriteCameralFile(*out, model, parallel.ModeNormalBP); e != nil
{
                return e
        }
        loaded, mode, e := entity.LoadCameral(*out)
        if e != nil {
                return e
        }
        if mode != parallel.ModeNormalBP {
                return fmt.Errorf("training mode changed during save")
        }
        restored, e := nativemetal.Mount(loaded)
        if e != nil {
                return e
        }
        defer restored.Close()
        want, e := s.Weights()
        if e != nil {
                return e
        }
        got, e := restored.Weights()
        if e != nil {
                return e
        }
        if !reflect.DeepEqual(want, got) {
                return fmt.Errorf("saved parameters changed on reload")
        }
        fmt.Printf("weights returned to Go; saved=%s; reload=exact\n", *out)
        return nil
}
```

Output

exit 0 · last run via `go run .`

```
Go parameters unchanged before SyncToGo
weights returned to Go;
saved=/Users/openfluke/gs/openfluke.github.io/welvet/examples/.cache/tmp/welvet-
apple-entity-689674332/trained.entity; reload=exact
```

Validate (harness)

```
cd welvet/examples/85-apple-save-weights && source ../env.sh && go run .
```

CHAPTER 86

Apple Metal — JSON backend placement

github.com/openfluke/welvet/nativemetal `verified Apple M5`

Why it exists

Choose which layers run on Apple GPU while retaining ordinary Go training entry points.

What it is

Enable the Metal router and apply JSON placement to a Stack: layer 0 on Go CPU, layers 1–2 on Metal.

Explicit placement

The example embeds placement.json; use `-config` to load another file. `nativemetal.EnableRouting()` installs the router for its owner lifetime. `placement.Stack` applies immediate-child numeric layer keys. Grid placement also uses sector keys such as `z,y,x,0` and local layer keys.

Routed execution maintains native sessions and returns updated parameters to Go automatically. Changing source weights remounts a session and resets transient native state. Automatic hardware/performance planning remains separate from explicit placement.

Requirements and library installation: chapter 82.

Go example

examples/86-apple-json-routing/main.go

Run: `cd welvet/examples/86-apple-json-routing && source ../env.sh && go run .`

```
// JSON selects backends; the existing Go Stack scheduler moves
activations/gradients.
package main
```

```
import (
        _ "embed"
        "flag"
        "fmt"
        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/dense"
        "github.com/openfluke/welvet/layers/parallel"
        "github.com/openfluke/welvet/nativecpp"
        "github.com/openfluke/welvet/nativemetal"
        "github.com/openfluke/welvet/quant"
        "github.com/openfluke/welvet/runtime/placement"
        "log"
        "os"
        "runtime"
)

//go:embed placement.json
var defaultPlan []byte

func main() {
        if e := run(); e != nil {
                log.Fatal(e)
        }
}
func run() error {
        path := flag.String("config", "", "optional replacement execution JSON")
        flag.Parse()
        if runtime.GOOS != "darwin" || runtime.GOARCH != "arm64" {
                fmt.Println("SKIP: Apple Metal needs Apple silicon macOS and cgo")
                return nil
        }
        raw := defaultPlan
        if *path != "" {
                var e error
                raw, e = os.ReadFile(*path)
                if e != nil {
                        return e
                }
        }
        cfg, e := nativecpp.ParseExecConfigJSON(raw)
        if e != nil {
                return e
        }
        router, e := nativemetal.EnableRouting()
        if e != nil {
                return e
        }
        defer router.Close()
        var ops []any
        for i := 0; i < 3; i++ {
                l, e := dense.NewConfigured(2, 2, core.ActivationTanh,
core.DTypeFloat32, quant.FormatNone, []float32{.8, .1, -.1, .9})
                if e != nil {
                        return e
                }
                ops = append(ops, l)
        }
        model, e := parallel.NewStack(ops...)
        if e != nil {
```

```
                return e
        }
        if e = placement.Stack(model, cfg, "main", "0,0,0,0"); e != nil {
                return e
        }
        for i, op := range ops {
                fmt.Printf("layer=%d backend=%s\n", i, op.
(*dense.Layer).Exec.Backend)
        }
        x := &core.Tensor[float32]{Shape: []int{1, 2}, Data: []float32{.5, -.3}}
        target := core.NewTensor[float32](1, 2)
        loss, e := parallel.TrainStackMSE(model, x, target, parallel.ModeNormalBP,
.03)
        if e != nil {
                return e
        }
        stats := router.Stats()
        if stats.Backwards == 0 || stats.Updates == 0 {
                return fmt.Errorf("plan did not execute native Metal backward/SGD")
        }
        fmt.Printf("loss=%.8g metal routing=%+v\nRouted updates are already in Go
stores; Close releases cached mounts.\n", loss, stats)
        return nil
}
```

Output

exit 0 · last run via `go run .`

```
layer=0 backend=cpu-tiled
layer=1 backend=metal
layer=2 backend=metal
loss=0.055711714 metal routing={Mounts:2 Forwards:4 Backwards:2 Updates:2
TrainingSteps:0}
Routed updates are already in Go stores; Close releases cached mounts.
```

Validate (harness)

```
cd welvet/examples/86-apple-json-routing && source ../env.sh && go run .
```

CHAPTER 87

Apple Metal — mixed Go/C++/Metal

github.com/openfluke/welvet/nativemetal `verified Apple M5`

Why it exists

Split a single model between the independent native engines and Go.

What it is

Run a 13-layer Dense Stack: Go layers 0–3, C++ layers 4–8 and Metal layers 9–12; verify actual native calls during training.

Three engines in one Go model

```
Go CPU [0–3] → welvet.cpp [4–8] → welvet.metal [9–12] → Go result
```

The embedded placement.json uses `go_cpu`, `native_cpu` and `metal`. Install both native libraries and use the normal cgo build. The example reports successful C++ C ABI calls and Metal routing counters after five NormalBP updates.

Activations and parameters cross backend boundaries. Live backend-specific Line, proxy, KV or rule state is not migrated between engines. Backend support still depends on the selected layer configuration.

Requirements and library installation: chapter 82.

Go example

examples/87-apple-mixed-backends/main.go

Run:

```
cd welvet/examples/87-apple-mixed-backends && source ../env.sh && go run .
```

```
// Keep a normal cgo build: -tags welvet_metal intentionally disables C++.
package main

import (
	_ "embed"
	"fmt"
	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/dense"
	"github.com/openfluke/welvet/layers/parallel"
	"github.com/openfluke/welvet/nativecpp"
	"github.com/openfluke/welvet/nativemetal"
	"github.com/openfluke/welvet/quant"
	"github.com/openfluke/welvet/runtime/placement"
	"log"
	"runtime"
)

//go:embed placement.json
var plan []byte

func main() {
	if e := run(); e != nil {
		log.Fatal(e)
	}
}
func run() error {
	if runtime.GOOS != "darwin" || runtime.GOARCH != "arm64" {
		fmt.Println("SKIP: mixed Apple routing needs Apple silicon macOS and
cgo")
		return nil
	}
	if !nativecpp.NativeCPUAvailable() {
		fmt.Println("SKIP: mixed routing needs the C++ artifact and normal
build without -tags welvet_metal")
		return nil
	}
	router, e := nativemetal.EnableRouting()
	if e != nil {
		return e
	}
	defer router.Close()
	cfg, e := nativecpp.ParseExecConfigJSON(plan)
	if e != nil {
		return e
	}
	var ops []any
	for i := 0; i < 13; i++ {
		l, e := dense.NewConfigured(2, 2, core.ActivationTanh,
core.DTypeFloat32, quant.FormatNone, []float32{.8, .12, -.15, .9})
		if e != nil {
			return e
		}
		ops = append(ops, l)
	}
	model, e := parallel.NewStack(ops...)
	if e != nil {
		return e
	}
	if e = placement.Stack(model, cfg, "main", "0,0,0,0"); e != nil {
```

```
                return e
        }
        x := &core.Tensor[float32]{Shape: []int{1, 2}, Data: []float32{.5, -.3}}
        target := core.NewTensor[float32](1, 2)
        before := nativecpp.CompletedCalls()
        var loss float64
        for i := 0; i < 5; i++ {
                loss, e = parallel.TrainStackMSE(model, x, target,
parallel.ModeNormalBP, .03)
                if e != nil {
                        return e
                }
        }
        calls := nativecpp.CompletedCalls() - before
        stats := router.Stats()
        if calls == 0 || stats.Backwards == 0 || stats.Updates == 0 {
                return fmt.Errorf("expected both C++ and Metal native calls")
        }
        fmt.Println("Go layers 0-3 -> C++ layers 4-8 -> Metal layers 9-12")
        fmt.Printf("loss=%.8g cpp calls=%d metal routing=%+v\n", loss, calls, stats)
        fmt.Println("Float32 activations travel forward; gradients travel back;
routed Metal weights return automatically to Go.")
        return nil
}
```

Output

exit 0 · last run via `go run .`

```
Go layers 0-3 -> C++ layers 4-8 -> Metal layers 9-12
loss=0.0043765833 cpp calls=75 metal routing={Mounts:4 Forwards:40 Backwards:20
Updates:20 TrainingSteps:0}
Float32 activations travel forward; gradients travel back; routed Metal weights
return automatically to Go.
```

Validate (harness)

```
cd welvet/examples/87-apple-mixed-backends && source ../env.sh && go run .
```

CHAPTER 88

Apple Metal — Mamba results and stress runner

captured snapshot

Why it exists

Measure real end-to-end Go bindings on sufficiently large models.

What it is

Display a labelled captured Mamba forward excerpt, then run the reproducible Go/C++/Metal stress matrix on your machine.

Captured forward excerpt

These user-supplied measurements were recorded on 2026-10-27. They are a historical snapshot, not a benchmark performed by loading this page. The five rows below are also printed by the example.

Dtype	Go ms	C++ ms	Metal ms	Metal speedup vs Go	Metal speedup vs C++
float64	89.314	236.242	28.959	3.084×	8.158×
float32	69.019	225.813	29.010	2.379×	7.784×
float16	190.545	1155.902	28.947	6.583×	39.932×
bfloat16	149.351	1049.180	29.003	5.149×	36.174×
fp8e4m3	265.882	1519.874	29.159	9.118×	52.124×

All 12 supplied forward rows reported Go/Metal exact-bit and tolerance agreement of 100%/100%. C++ unsigned storage differed in exact bits (uint64 21.2%, uint32 45.4%) while tolerance agreement remained 100%. This is numerical agreement, not prediction accuracy or a guarantee for future training runs.

Run the matrix

The bundled stress runner documentation describes all metrics and the full excerpt. It compares Mamba across 34 parameter dtypes and 29 training modes: 34 forward rows plus 986 training rows, or 1,020 rows, each covering three backends. Use the normal cgo build with both native libraries installed.

```
cd welvet/examples/apple_stress_test
./run.sh -profile smoke -samples 1 -iters 1 -steps 8
./run.sh
```

The default stress profile uses DModel 512, batch 8, sequence 32, three samples, three iterations, eight training steps and one warm-up. Reports include median wall time, setup and final weight-pull costs, speed ratios, time saved, exact-bit and tolerance agreement, and repeatability. CSV/JSONL/summary JSON are written progressively under results/.

Small-shape verification covered the full matrix; a separate stress-size Float32 NormalBP check measured Metal gains. That evidence does not claim the full large-shape matrix has already been timed. CPU/GPU numerical differences can grow over long training horizons. Best effort aims at exact agreement; project acceptance labels use 60/80/100 tolerance bands, not percentages of correct tensor elements or model accuracy.

Requirements and library installation: chapter 82.

Go example

examples/88-apple-mamba-results/main.go

Run: `cd welvet/examples/88-apple-mamba-results && source ../env.sh && go run .`

```
// A labelled snapshot of user-provided forward results; no GPU run is implied.
package main

import "fmt"

func main() {
        fmt.Println("Captured Mamba forward excerpt, supplied 2026-10-27 (not a live
benchmark)")
        fmt.Printf("%-10s %12s %12s %12s %13s %14s\n", "dtype", "Go ms", "C++ ms",
"Metal ms", "Metal/Go x", "Metal/C++ x")
        rows := []struct {
                dtype               string
```

```
                goUS, cppUS, metalUS float64
        }{
                {"float64", 89314.5, 236241.7, 28958.6}, {"float32", 69018.5,
225812.9, 29010.4},
                {"float16", 190545.0, 1155902.2, 28946.8}, {"bfloat16", 149351.3,
1049179.8, 29003.3},
                {"fp8e4m3", 265881.6, 1519874.3, 29159.0},
        }
        for _, r := range rows {
                fmt.Printf("%-10s %12.4f %12.4f %12.4f %13.3f %14.3f\n", r.dtype,
r.goUS/1000, r.cppUS/1000, r.metalUS/1000, r.goUS/r.metalUS, r.cppUS/r.metalUS)
        }
        fmt.Println("Go/Metal exact bits and within-tolerance agreement: 100%/100%
across all 12 supplied forward rows.")
        fmt.Println("Full excerpt and metric definitions:
../apple_stress_test/README.md")
        fmt.Println("Run your own 34-dtype / 29-mode matrix: cd ../apple_stress_test
&& ./run.sh")
        fmt.Println("Small verification: ./run.sh -profile smoke -samples 1 -iters 1
-steps 8")
}
```

Output

exit 0 · last run via `go run .`

```
Captured Mamba forward excerpt, supplied 2026-10-27 (not a live benchmark)
dtype             Go ms       C++ ms     Metal ms    Metal/Go x    Metal/C++ x
float64         89.3145     236.2417      28.9586         3.084          8.158
float32         69.0185     225.8129      29.0104         2.379          7.784
float16        190.5450    1155.9022      28.9468         6.583         39.932
bfloat16       149.3513    1049.1798      29.0033         5.149         36.174
fp8e4m3        265.8816    1519.8743      29.1590         9.118         52.124
Go/Metal exact bits and within-tolerance agreement: 100%/100% across all 12 supplied
forward rows.
Full excerpt and metric definitions: ../apple_stress_test/README.md
Run your own 34-dtype / 29-mode matrix: cd ../apple_stress_test && ./run.sh
Small verification: ./run.sh -profile smoke -samples 1 -iters 1 -steps 8
```

Validate (harness)

```
cd welvet/examples/88-apple-mamba-results && source ../env.sh && go run .
```

CHAPTER 89

NVIDIA CUDA — mount and forward

github.com/openfluke/welvet/nativecuda `linux/windows amd64`

Why it exists

Run a Go layer on NVIDIA GPU with an explicit lifetime.

What it is

Mount an identity Dense layer, inspect the CUDA version and device identity, and forward a Float32 tensor.

Session ownership

Create a session with `nativecuda.Mount`, call `Forward` repeatedly and close it when finished. Mount accepts supported layers, Stack, Parallel and Grid layouts. The library is independent of welvet.cpp and welvet.metal.

Requirements and backend boundaries

Go Welvet **v1.4.0** uses the independent **welvet.cuda 0.79** library (mounted model ABI 1). Install from its checkout:

```
cd ../welvet.cuda
./scripts/linux_install_go.sh
# Windows amd64 (MSVC + CUDA Toolkit):
./scripts/windows_install_go.ps1
```

Execution requires **Linux x86_64 or Windows amd64**, Go with cgo, and an NVIDIA driver/runtime. Set `WELVET_CUDA_LIBRARY` to override library discovery. On Windows, ensure the CUDA runtime (`cudart`) resolves via `PATH`. Ordinary Go CPU use does not require an NVIDIA GPU. Unsupported hosts print SKIP; missing libraries or no GPU on a supported host also soft-SKIP in these book examples.

Go schedules work and manages memory; numerical CUDA forward, backward and weight updates execute device kernels. Activations are Float32; FormatNone parameter storage follows the CUDA dtype matrix. Small examples demonstrate APIs; GPU launch costs can outweigh useful work at small shapes.

`cpu-tiled` / `go_cpu` means Go CPU; `native_cpu` means welvet.cpp; `cuda` means the independent NVIDIA GPU library. Examples 89–93 can use `go run -tags welvet_cuda .` to exclude C++ linkage. Chapter 94 requires the normal cgo build for both native engines. JSON routing (§93–§94) is **Linux amd64** today; explicit Mount works on Windows amd64 as well.

Go example

examples/89-nvidia-cuda/main.go

Run: `cd welvet/examples/89-nvidia-cuda && source ../env.sh && go run .`

```
// Mount one layer, execute its forward pass on NVIDIA CUDA, and release it.
package main

import (
	"flag"
	"fmt"
	"log"
	"runtime"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/dense"
	"github.com/openfluke/welvet/nativecuda"
	"github.com/openfluke/welvet/quant"
)

func main() {
	if e := run(); e != nil {
		log.Fatal(e)
	}
}

func run() error {
	library := flag.String("library", "", "optional libwelvet_cuda.so/.dll path")
	flag.Parse()
	if !cudaPlatformOK() {
		fmt.Println("SKIP: NVIDIA CUDA needs linux/amd64 or windows/amd64 and
cgo")
		return nil
	}
	l, e := dense.NewConfigured(2, 2, core.ActivationLinear, core.DTypeFloat32,
quant.FormatNone, []float32{1, 0, 0, 1})
	if e != nil {
		return e
```

```
        }
        session, e := nativecuda.Mount(l, nativecuda.Options{LibraryPath: *library})
        if e != nil {
                fmt.Printf("SKIP: CUDA unavailable (%v)\n", e)
                return nil
        }
        defer session.Close()
        x := &core.Tensor[float32]{Shape: []int{1, 2}, Data: []float32{.5, -.25}}
        post, e := session.Forward(x)
        if e != nil {
                return e
        }
        fmt.Printf("backend=cuda version=%s\ndevice=%s\ninput=%v output=%v
shape=%v\n", session.Version(), session.Identity(), x.Data, post.Data, post.Shape)
        return nil
}

func cudaPlatformOK() bool {
        return runtime.GOARCH == "amd64" && (runtime.GOOS == "linux" || runtime.GOOS
== "windows")
}
```

Output

exit 0 · last run via `go run .`

```
SKIP: NVIDIA CUDA needs linux/amd64 or windows/amd64 and cgo
```

Validate (harness)

```
cd welvet/examples/89-nvidia-cuda && source ../env.sh && go run .
```

CHAPTER 90

NVIDIA CUDA — Mamba parameter dtypes

github.com/openfluke/welvet/nativecuda `linux/windows amd64`

Why it exists

Exercise native storage formats without changing the activation interface.

What it is

Run a seeded Mamba forward pass across all 34 native parameter dtypes, or select one with -dtype.

Storage versus arithmetic

`-dtype all` sweeps native parameter representations. The CUDA execution interface uses Float32 activations. Quantized FormatNone dtype storage and packed quantization formats are different coverage claims.

```
go run . -dtype all
go run . -dtype float16
```

Requirements and library installation (Linux + Windows): chapter 89.

Go example

examples/90-nvidia-mamba-dtypes/main.go

Run: `cd welvet/examples/90-nvidia-mamba-dtypes && source ../env.sh && go run .`

```
// Float32 tensors with independently selected native parameter encodings.
package main

import (
        "flag"
```

```
        "fmt"
        "log"
        "runtime"
        "strings"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/mamba"
        "github.com/openfluke/welvet/nativecuda"
        "github.com/openfluke/welvet/quant"
)

func main() {
        if e := run(); e != nil {
                log.Fatal(e)
        }
}

func run() error {
        selected := flag.String("dtype", "all", "native parameter dtype name, or all
34")
        flag.Parse()
        if !cudaPlatformOK() {
                fmt.Println("SKIP: NVIDIA CUDA needs linux/amd64 or windows/amd64 and
cgo")
                return nil
        }
        count := 0
        for _, dt := range core.AllDTypes {
                if *selected != "all" && !strings.EqualFold(*selected, dt.String()) {
                        continue
                }
                if e := forward(dt); e != nil {
                        if count == 0 && *selected == "all" {
                                fmt.Printf("SKIP: CUDA unavailable (%v)\n", e)
                                return nil
                        }
                        return fmt.Errorf("%s: %w", dt, e)
                }
                count++
        }
        if count == 0 {
                return fmt.Errorf("unknown dtype %q", *selected)
        }
        fmt.Printf("Mamba dtypes=%d; activations=float32; parameters=FormatNone
native dtype\n", count)
        return nil
}

func forward(dt core.DType) error {
        const d, inner = 8, 16
        in, out := make([]float32, 2*inner*d), make([]float32, d*inner)
        for i := range in {
                in[i] = float32(i%17-8) * .01
        }
        for i := range out {
                out[i] = float32(i%13-6) * .01
        }
        model, e := mamba.NewConfigured(mamba.Config{DModel: d, DState: 4, Expand: 2,
SeqLen: 3}, dt, quant.FormatNone, in, out, nil, nil)
        if e != nil {
```

```
                return e
        }
        s, e := nativecuda.Mount(model)
        if e != nil {
                return e
        }
        defer s.Close()
        x := core.NewTensor[float32](1, 3, d)
        for i := range x.Data {
                x.Data[i] = float32(i%7-3) * .05
        }
        post, e := s.Forward(x)
        if e != nil {
                return e
        }
        var checksum float64
        for _, v := range post.Data {
                checksum += float64(v)
        }
        fmt.Printf("%-10s shape=%v checksum=%.8g\n", dt, post.Shape, checksum)
        return nil
}

func cudaPlatformOK() bool {
        return runtime.GOARCH == "amd64" && (runtime.GOOS == "linux" || runtime.GOOS
== "windows")
}
```

Output

exit 0 · last run via `go run .`

```
SKIP: NVIDIA CUDA needs linux/amd64 or windows/amd64 and cgo
```

Validate (harness)

```
cd welvet/examples/90-nvidia-mamba-dtypes && source ../env.sh && go run .
```

CHAPTER 91

NVIDIA CUDA — training modes

github.com/openfluke/welvet/nativecuda `linux/windows amd64`

Why it exists

Keep repeated training updates in the mounted CUDA session.

What it is

Train two Dense layers using one of the 29 numbered modes, with persistent Step Line state and real Grid layouts for Mesh modes.

Training families

`-mode all` exercises all 29 modes. Step modes retain their Line session across calls. Mesh modes use a three-cell Grid with disabled ingress and two active layers, advancing four ticks per call. Bundled **welvet.cuda 0.79** covers train modes through StepTweenSplitSparse. These small demonstrations validate usage; they are not performance measurements.

```
go run . -mode NormalBP -steps 8
go run . -mode all -steps 8
```

Explicit mounted sessions keep updated weights native until `SyncToGo()`. Close discards any updates not returned to Go. See chapter 92 before saving.

Requirements: chapter 89.

Go example

examples/91-nvidia-training/main.go

Run: `cd welvet/examples/91-nvidia-training && source ../env.sh && go run .`

```
// Native CUDA MSE training: persistent Stack Line state or an actual Grid clock.
package main

import (
	"flag"
	"fmt"
	"log"
	"runtime"

	"github.com/openfluke/welvet/architecture"
	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/dense"
	"github.com/openfluke/welvet/layers/parallel"
	"github.com/openfluke/welvet/nativecuda"
	"github.com/openfluke/welvet/quant"
)

func main() {
	if e := run(); e != nil {
		log.Fatal(e)
	}
}

func run() error {
	selected := flag.String("mode", "NormalBP", "training mode name, or all 29")
	steps := flag.Int("steps", 8, "training calls; retain the session for Step
pipeline state")
	flag.Parse()
	if *steps < 1 {
		return fmt.Errorf("steps must be positive")
	}
	if !cudaPlatformOK() {
		fmt.Println("SKIP: NVIDIA CUDA needs linux/amd64 or windows/amd64 and
cgo")
		return nil
	}
	if *selected == "all" {
		for mode := parallel.ModeNormalBP; mode <=
parallel.ModeStepTweenSplitSparse; mode++ {
			if e := train(mode, *steps); e != nil {
				if mode == parallel.ModeNormalBP {
					fmt.Printf("SKIP: CUDA unavailable (%v)\n",
e)
					return nil
				}
				return e
			}
		}
		return nil
	}
	mode, e := parallel.ParseTrainMode(*selected)
	if e != nil {
		return e
	}
	if mode < parallel.ModeNormalBP || mode > parallel.ModeStepTweenSplitSparse {
		return fmt.Errorf("choose one of the 29 update modes")
	}
	if e := train(mode, *steps); e != nil {
		fmt.Printf("SKIP: CUDA unavailable (%v)\n", e)
```

```
                return nil
        }
        return nil
}

func train(mode parallel.TrainMode, steps int) error {
        var ops []any
        for layer := 0; layer < 2; layer++ {
                w := make([]float32, 16)
                for i := range w {
                        w[i] = float32(i%5-2) * .04
                }
                l, e := dense.NewConfigured(4, 4, core.ActivationTanh,
core.DTypeFloat32, quant.FormatNone, w)
                if e != nil {
                        return e
                }
                ops = append(ops, l)
        }
        stack, e := parallel.NewStack(ops...)
        if e != nil {
                return e
        }
        var model any = stack
        schedule := "Stack"
        if mode.RequiresGrid() {
                g := architecture.NewGrid(3, 1, 1, 1)
                g.Cells[0].Layer.IsDisabled = true
                for i, op := range ops {
                        g.Cells[i+1].Layer = op.(*dense.Layer).Core
                        g.Cells[i+1].Op = op
                }
                model = g
                schedule = "Grid mesh clock (4 ticks/call)"
        } else if mode.IsLineStep() {
                schedule = "Stack Line pipeline"
        }
        s, e := nativecuda.Mount(model)
        if e != nil {
                return e
        }
        defer s.Close()
        x := core.NewTensor[float32](2, 4)
        for i := range x.Data {
                x.Data[i] = float32(i%5-2) * .1
        }
        target := core.NewTensor[float32](2, 4)
        var loss float64
        for i := 0; i < steps; i++ {
                loss, e = s.TrainMSE(x, target, mode, .01, 4)
                if e != nil {
                        return fmt.Errorf("%s: %w", mode, e)
                }
        }
        if e = s.SyncToGo(); e != nil {
                return e
        }
        fmt.Printf("mode=%s schedule=%s calls=%d loss=%.8g weights=synced-to-Go\n",
mode, schedule, steps, loss)
        return nil
```

```
}

func cudaPlatformOK() bool {
        return runtime.GOARCH == "amd64" && (runtime.GOOS == "linux" || runtime.GOOS
== "windows")
}
```

Output

exit 0 · last run via `go run .`

```
SKIP: NVIDIA CUDA needs linux/amd64 or windows/amd64 and cgo
```

Validate (harness)

```
cd welvet/examples/91-nvidia-training && source ../env.sh && go run .
```

CHAPTER 92

NVIDIA CUDA — weight return and ENTITY save

github.com/openfluke/welvet/nativecuda

linux/windows amd64

Why it exists

Return trained weights to Go before executing or saving the Go model.

What it is

Train natively, explicitly SyncToGo, save an ENTITY, reload it and compare the native parameter snapshots.

Weight handoff

```
session, err := nativecuda.Mount(layer)
// Check err; defer session.Close().
// Repeat session.TrainMSE(...) as needed.
err = session.SyncToGo()
// Check err, then save through Go ENTITY IO.
```

Mount copies model state into a native session. The original Go parameters stay unchanged while that explicit session trains. `SyncToGo()` returns parameter bytes, scales and Float64 biases for Go saving. This saves model parameters, not a portable checkpoint of live Line, proxy, KV or rule state.

Use `-out /path/model.entity` to choose the output location. The default uses a temporary directory and prints it.

Requirements: chapter 89.

Go example

examples/92-nvidia-save-weights/main.go

Run: `cd welvet/examples/92-nvidia-save-weights && source ../env.sh && go run .`

```
// Explicit mount ownership, weight return to Go, ENTITY save and reload.
package main

import (
	"flag"
	"fmt"
	"log"
	"os"
	"path/filepath"
	"reflect"
	"runtime"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/dense"
	"github.com/openfluke/welvet/layers/parallel"
	"github.com/openfluke/welvet/model/entity"
	"github.com/openfluke/welvet/nativecuda"
	"github.com/openfluke/welvet/quant"
)

func main() {
	if e := run(); e != nil {
		log.Fatal(e)
	}
}

func run() error {
	out := flag.String("out", "", "ENTITY destination; default is a new temporary
directory")
	flag.Parse()
	if !cudaPlatformOK() {
		fmt.Println("SKIP: NVIDIA CUDA needs linux/amd64 or windows/amd64 and
cgo")
		return nil
	}
	l, e := dense.NewConfigured(2, 2, core.ActivationLinear, core.DTypeFloat32,
quant.FormatNone, []float32{.3, -.2, .1, .4})
	if e != nil {
		return e
	}
	model, e := parallel.NewStack(l)
	if e != nil {
		return e
	}
	before, e := l.Weights.FlattenF32()
	if e != nil {
		return e
	}
	s, e := nativecuda.Mount(model)
	if e != nil {
		fmt.Printf("SKIP: CUDA unavailable (%v)\n", e)
		return nil
	}
	defer s.Close()
	x := &core.Tensor[float32]{Shape: []int{1, 2}, Data: []float32{.5, .25}}
	target := core.NewTensor[float32](1, 2)
```

```
        for i := 0; i < 5; i++ {
                if _, e = s.TrainMSE(x, target, parallel.ModeNormalBP, .1); e != nil
{
                        return e
                }
        }
        unchanged, e := l.Weights.FlattenF32()
        if e != nil {
                return e
        }
        if !reflect.DeepEqual(before, unchanged) {
                return fmt.Errorf("explicit mount unexpectedly changed Go parameters
before SyncToGo")
        }
        fmt.Println("Go parameters unchanged before SyncToGo")
        if e = s.SyncToGo(); e != nil {
                return e
        }
        after, e := l.Weights.FlattenF32()
        if e != nil {
                return e
        }
        if reflect.DeepEqual(before, after) {
                return fmt.Errorf("expected trained parameters after SyncToGo")
        }
        if *out == "" {
                dir, e := os.MkdirTemp("", "welvet-cuda-entity-")
                if e != nil {
                        return e
                }
                *out = filepath.Join(dir, "trained.entity")
        }
        if e = entity.WriteCameralFile(*out, model, parallel.ModeNormalBP); e != nil
{
                return e
        }
        loaded, mode, e := entity.LoadCameral(*out)
        if e != nil {
                return e
        }
        if mode != parallel.ModeNormalBP {
                return fmt.Errorf("training mode changed during save")
        }
        restored, e := nativecuda.Mount(loaded)
        if e != nil {
                return e
        }
        defer restored.Close()
        want, e := s.Weights()
        if e != nil {
                return e
        }
        got, e := restored.Weights()
        if e != nil {
                return e
        }
        if !reflect.DeepEqual(want, got) {
                return fmt.Errorf("saved parameters changed on reload")
        }
        fmt.Printf("weights returned to Go; saved=%s; reload=exact\n", *out)
```

```
        return nil
}

func cudaPlatformOK() bool {
        return runtime.GOARCH == "amd64" && (runtime.GOOS == "linux" || runtime.GOOS
== "windows")
}
```

Output

exit 0 · last run via `go run .`

```
SKIP: NVIDIA CUDA needs linux/amd64 or windows/amd64 and cgo
```

Validate (harness)

```
cd welvet/examples/92-nvidia-save-weights && source ../env.sh && go run .
```

CHAPTER 93

NVIDIA CUDA — JSON backend placement

github.com/openfluke/welvet/nativecuda `linux amd64 routing`

Why it exists

Choose which layers run on NVIDIA GPU while retaining ordinary Go training entry points.

What it is

Enable the CUDA router and apply JSON placement to a Stack: layer 0 on Go CPU, layers 1–2 on CUDA.

Explicit placement

The example embeds placement.json; use `-config` to load another file. `nativecuda.EnableRouting()` installs the router for its owner lifetime (**Linux amd64**). `placement.Stack` applies immediate-child numeric layer keys.

Routed execution maintains native sessions and returns updated parameters to Go automatically. Changing source weights remounts a session and resets transient native state. Automatic hardware/performance planning remains separate from explicit placement.

Windows amd64 supports explicit `Mount` (chapters 89–92); automatic JSON routing is Linux-only in this release.

Requirements: chapter 89.

Go example

examples/93-nvidia-json-routing/main.go

Run:

```
cd welvet/examples/93-nvidia-json-routing && source ../env.sh && go run .
```

```
// JSON selects backends; the existing Go Stack scheduler moves
activations/gradients.
package main

import (
        _ "embed"
        "flag"
        "fmt"
        "log"
        "os"
        "runtime"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/dense"
        "github.com/openfluke/welvet/layers/parallel"
        "github.com/openfluke/welvet/nativecpp"
        "github.com/openfluke/welvet/nativecuda"
        "github.com/openfluke/welvet/quant"
        "github.com/openfluke/welvet/runtime/placement"
)

//go:embed placement.json
var defaultPlan []byte

func main() {
        if e := run(); e != nil {
                log.Fatal(e)
        }
}

func run() error {
        path := flag.String("config", "", "optional replacement execution JSON")
        flag.Parse()
        if runtime.GOOS != "linux" || runtime.GOARCH != "amd64" {
                fmt.Println("SKIP: CUDA JSON routing needs Linux amd64 and cgo")
                return nil
        }
        raw := defaultPlan
        if *path != "" {
                var e error
                raw, e = os.ReadFile(*path)
                if e != nil {
                        return e
                }
        }
        cfg, e := nativecpp.ParseExecConfigJSON(raw)
        if e != nil {
                return e
        }
        router, e := nativecuda.EnableRouting()
        if e != nil {
                fmt.Printf("SKIP: CUDA routing unavailable (%v)\n", e)
                return nil
        }
        defer router.Close()
        var ops []any
        for i := 0; i < 3; i++ {
                l, e := dense.NewConfigured(2, 2, core.ActivationTanh,
core.DTypeFloat32, quant.FormatNone, []float32{.8, .1, -.1, .9})
```

```
                if e != nil {
                        return e
                }
                ops = append(ops, l)
        }
        model, e := parallel.NewStack(ops...)
        if e != nil {
                return e
        }
        if e = placement.Stack(model, cfg, "main", "0,0,0,0"); e != nil {
                return e
        }
        for i, op := range ops {
                fmt.Printf("layer=%d backend=%s\n", i, op.
(*dense.Layer).Exec.Backend)
        }
        x := &core.Tensor[float32]{Shape: []int{1, 2}, Data: []float32{.5, -.3}}
        target := core.NewTensor[float32](1, 2)
        loss, e := parallel.TrainStackMSE(model, x, target, parallel.ModeNormalBP,
.03)
        if e != nil {
                fmt.Printf("SKIP: CUDA train unavailable (%v)\n", e)
                return nil
        }
        stats := router.Stats()
        if stats.Backwards == 0 || stats.Updates == 0 {
                return fmt.Errorf("plan did not execute native CUDA backward/SGD")
        }
        fmt.Printf("loss=%.8g cuda routing=%+v\nRouted updates are already in Go
stores; Close releases cached mounts.\n", loss, stats)
        return nil
}
```

Output

exit 0 · last run via `go run .`

```
SKIP: CUDA JSON routing needs Linux amd64 and cgo
```

Validate (harness)

```
cd welvet/examples/93-nvidia-json-routing && source ../env.sh && go run .
```

CHAPTER 94

NVIDIA CUDA — mixed Go/C++/CUDA

github.com/openfluke/welvet/nativecuda

linux amd64 routing

Why it exists

Split a single model between the independent native engines and Go.

What it is

Run a 13-layer Dense Stack: Go layers 0–3, C++ layers 4–8 and CUDA layers 9–12; verify actual native calls during training.

Three engines in one Go model

```
Go CPU [0–3] → welvet.cpp [4–8] → welvet.cuda [9–12] → Go result
```

The embedded placement.json uses `go_cpu`, `native_cpu` and `cuda`. Install both native libraries and use the normal cgo build (do **not** pass `-tags welvet_cuda`). The example reports successful C++ C ABI calls and CUDA routing counters after five NormalBP updates.

Activations and parameters cross backend boundaries. Live backend-specific Line, proxy, KV or rule state is not migrated between engines.

Requirements: Linux amd64 + chapter 89.

Go example

examples/94-nvidia-mixed-backends/main.go

Run:

```
cd welvet/examples/94-nvidia-mixed-backends && source ../env.sh && go run .
```

```
// Keep a normal cgo build: -tags welvet_cuda intentionally disables C++.
package main

import (
	_ "embed"
	"fmt"
	"log"
	"runtime"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/dense"
	"github.com/openfluke/welvet/layers/parallel"
	"github.com/openfluke/welvet/nativecpp"
	"github.com/openfluke/welvet/nativecuda"
	"github.com/openfluke/welvet/quant"
	"github.com/openfluke/welvet/runtime/placement"
)

//go:embed placement.json
var plan []byte

func main() {
	if e := run(); e != nil {
		log.Fatal(e)
	}
}

func run() error {
	if runtime.GOOS != "linux" || runtime.GOARCH != "amd64" {
		fmt.Println("SKIP: mixed CUDA routing needs Linux amd64 and cgo")
		return nil
	}
	if !nativecpp.NativeCPUAvailable() {
		fmt.Println("SKIP: mixed routing needs the C++ artifact and normal
build without -tags welvet_cuda")
		return nil
	}
	router, e := nativecuda.EnableRouting()
	if e != nil {
		fmt.Printf("SKIP: CUDA routing unavailable (%v)\n", e)
		return nil
	}
	defer router.Close()
	cfg, e := nativecpp.ParseExecConfigJSON(plan)
	if e != nil {
		return e
	}
	var ops []any
	for i := 0; i < 13; i++ {
		l, e := dense.NewConfigured(2, 2, core.ActivationTanh,
core.DTypeFloat32, quant.FormatNone, []float32{.8, .12, -.15, .9})
		if e != nil {
			return e
		}
		ops = append(ops, l)
	}
	model, e := parallel.NewStack(ops...)
	if e != nil {
		return e
```

```
        }
        if e = placement.Stack(model, cfg, "main", "0,0,0,0"); e != nil {
                return e
        }
        x := &core.Tensor[float32]{Shape: []int{1, 2}, Data: []float32{.5, -.3}}
        target := core.NewTensor[float32](1, 2)
        before := nativecpp.CompletedCalls()
        var loss float64
        for i := 0; i < 5; i++ {
                loss, e = parallel.TrainStackMSE(model, x, target,
parallel.ModeNormalBP, .03)
                if e != nil {
                        fmt.Printf("SKIP: mixed train unavailable (%v)\n", e)
                        return nil
                }
        }
        calls := nativecpp.CompletedCalls() - before
        stats := router.Stats()
        if calls == 0 || stats.Backwards == 0 || stats.Updates == 0 {
                return fmt.Errorf("expected both C++ and CUDA native calls")
        }
        fmt.Println("Go layers 0-3 -> C++ layers 4-8 -> CUDA layers 9-12")
        fmt.Printf("loss=%.8g cpp calls=%d cuda routing=%+v\n", loss, calls, stats)
        fmt.Println("Float32 activations travel forward; gradients travel back;
routed CUDA weights return automatically to Go.")
        return nil
}
```

Output

exit 0 · last run via `go run .`

```
SKIP: mixed CUDA routing needs Linux amd64 and cgo
```

Validate (harness)

```
cd welvet/examples/94-nvidia-mixed-backends && source ../env.sh && go run .
```

CHAPTER 95

NVIDIA CUDA — results and suite pointers

host-only pointers

Why it exists

Find live CUDA parity/timing matrices after the small usage chapters.

What it is

Print Go v1.4.0 / welvet.cuda 0.79 pins and commands for w2a go_v_cuda and cuda_smoke. No GPU required.

Where live results live

This chapter is host-only. It does not claim timings from loading the page. Run the sibling matrices on a machine with an NVIDIA GPU:

```
cd ../../w2a && go test ./tests/go_v_cuda -count=1 -timeout 180m
cd ../../welvet.cuda/examples/cuda_smoke && go run .
cd welvet/examples && ./run_nvidia.sh
```

Engine pin: **welvet.cuda 0.79**, model ABI **1**, Go binding `nativecuda` (Welvet **v1.4.0**). Platforms: Linux x86_64 + Windows amd64 for Mount; Linux amd64 for EnableRouting.

Requirements overview: chapter 89.

Go example

examples/95-nvidia-cuda-results/main.go

Run: `cd welvet/examples/95-nvidia-cuda-results && source ../env.sh && go run .`

```
// Pointers to CUDA validation matrices; no GPU run is implied.
package main
```

```
import "fmt"

func main() {
        fmt.Println("NVIDIA CUDA results pointers (not a live benchmark)")
        fmt.Println()
        fmt.Println("Engine pin: welvet.cuda 0.79 · model ABI 1 · Go binding:
nativecuda (Welvet v1.4.0)")
        fmt.Println("Usage tour: 89 mount · 90 Mamba dtypes · 91 train modes · 92
SyncToGo/ENTITY · 93 JSON · 94 mixed")
        fmt.Println()
        fmt.Println("Live matrices (sibling checkouts):")
        fmt.Println("  cd ../../w2a && go test ./tests/go_v_cuda -count=1 -timeout
180m")
        fmt.Println("  cd ../../welvet.cuda/examples/cuda_smoke && go run .")
        fmt.Println()
        fmt.Println("Install shared library into Go:")
        fmt.Println("  cd ../../welvet.cuda && ./scripts/linux_install_go.sh")
        fmt.Println()
        fmt.Println("Small usage tour without the large matrix:")
        fmt.Println("  ../run_nvidia.sh")
}
```

Output

exit 0 · last run via `go run .`

```
NVIDIA CUDA results pointers (not a live benchmark)

Engine pin: welvet.cuda 0.79 · model ABI 1 · Go binding: nativecuda (Welvet v1.4.0)
Usage tour: 89 mount · 90 Mamba dtypes · 91 train modes · 92 SyncToGo/ENTITY · 93
JSON · 94 mixed

Live matrices (sibling checkouts):
  cd ../../w2a && go test ./tests/go_v_cuda -count=1 -timeout 180m
  cd ../../welvet.cuda/examples/cuda_smoke && go run .

Install shared library into Go:
  cd ../../welvet.cuda && ./scripts/linux_install_go.sh

Small usage tour without the large matrix:
  ../run_nvidia.sh
```

Validate (harness)

```
cd welvet/examples/95-nvidia-cuda-results && source ../env.sh && go run .
```

CHAPTER 96

AMD ROCm — mount and forward

github.com/openfluke/welvet/nativerocm `linux/windows amd64`

Why it exists

Run a Go layer on an AMD GPU with an explicit lifetime via the independent welvet.rocm library.

What it is

Mount an identity Dense layer, inspect the ROCm version and device identity, and forward a Float32 tensor.

Session ownership

Create a session with `nativerocm.Mount`, call `Forward` repeatedly and close it when finished. Mount accepts supported layers, Stack, Parallel and Grid layouts. The library is independent of welvet.cpp, welvet.metal and welvet.cuda.

Requirements and backend boundaries

Go Welvet **v1.5.0** adds the independent **welvet.rocm 0.79** library (mounted model ABI 1) at github.com/openfluke/welvet.rocm. Install from its checkout:

```
cd ../welvet.rocm
./scripts/linux_install_go.sh
# Windows amd64 (MSVC + ROCm / HIP):
./scripts/windows_install_go.ps1
```

Execution requires **Linux x86_64 or Windows amd64**, Go with cgo, and an AMD driver/runtime with HIP (`libamdhip64`). Set `WELVET_ROCM_LIBRARY` to override library discovery. Ordinary Go CPU use does not require an AMD GPU. Unsupported hosts print SKIP; missing libraries or no GPU on a supported host also soft-SKIP in these book examples.

Honest coverage: library build/load is validated; **GPU compute remains untested** until a ROCm-capable HSA device is available (Polaris/Picasso-class hosts do not qualify).

Go schedules work and manages memory; numerical ROCm forward, backward and weight updates execute device kernels when HSA is present. Activations are Float32; FormatNone parameter storage follows the ROCm dtype matrix. Small examples demonstrate APIs; GPU launch costs can outweigh useful work at small shapes.

`cpu-tiled` / `go_cpu` means Go CPU; `native_cpu` means welvet.cpp; `rocm` means the independent AMD GPU library. Examples 96–100 can use `go run -tags welvet_rocm .` to exclude C++ linkage. Chapter 101 requires the normal cgo build for both native engines. JSON routing (§100–§101) is **Linux amd64** today; explicit Mount works on Windows amd64 as well.

Go example

examples/96-amd-rocm/main.go

Run: `cd welvet/examples/96-amd-rocm && source ../env.sh && go run .`

```
// Mount one layer, execute its forward pass on AMD ROCm, and release it.
package main

import (
        "flag"
        "fmt"
        "log"
        "runtime"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/dense"
        "github.com/openfluke/welvet/nativerocm"
        "github.com/openfluke/welvet/quant"
)

func main() {
        if e := run(); e != nil {
                log.Fatal(e)
        }
}

func run() error {
        library := flag.String("library", "", "optional libwelvet_rocm.so/.dll path")
        flag.Parse()
        if !rocmPlatformOK() {
                fmt.Println("SKIP: AMD ROCm needs linux/amd64 or windows/amd64 and
cgo")
                return nil
        }
```

```
	l, e := dense.NewConfigured(2, 2, core.ActivationLinear, core.DTypeFloat32,
quant.FormatNone, []float32{1, 0, 0, 1})
	if e != nil {
		return e
	}
	session, e := nativerocm.Mount(l, nativerocm.Options{LibraryPath: *library})
	if e != nil {
		fmt.Printf("SKIP: ROCm unavailable (%v)\n", e)
		return nil
	}
	defer session.Close()
	x := &core.Tensor[float32]{Shape: []int{1, 2}, Data: []float32{.5, -.25}}
	post, e := session.Forward(x)
	if e != nil {
		return e
	}
	fmt.Printf("backend=rocm version=%s\ndevice=%s\ninput=%v output=%v
shape=%v\n", session.Version(), session.Identity(), x.Data, post.Data, post.Shape)
	return nil
}

func rocmPlatformOK() bool {
	return runtime.GOARCH == "amd64" && (runtime.GOOS == "linux" || runtime.GOOS
== "windows")
}
```

Output

exit 0 · last run via `go run .`

```
SKIP: AMD ROCm needs linux/amd64 or windows/amd64 and cgo
```

Validate (harness)

```
cd welvet/examples/96-amd-rocm && source ../env.sh && go run .
```

CHAPTER 97

AMD ROCm — Mamba parameter dtypes

github.com/openfluke/welvet/nativerocm `linux/windows amd64`

Why it exists

Exercise native storage formats without changing the activation interface.

What it is

Run a seeded Mamba forward pass across all 34 native parameter dtypes, or select one with -dtype.

Storage versus arithmetic

`-dtype all` sweeps native parameter representations. The ROCm execution interface uses Float32 activations. Quantized FormatNone dtype storage and packed quantization formats are different coverage claims.

```
go run . -dtype all
go run . -dtype float16
```

Requirements and library installation (Linux + Windows): chapter 96. GPU compute untested until a ROCm-capable HSA device is available.

Go example

examples/97-amd-mamba-dtypes/main.go

Run: `cd welvet/examples/97-amd-mamba-dtypes && source ../env.sh && go run .`

```
// Float32 tensors with independently selected native parameter encodings.
package main

import (
```

```
        "flag"
        "fmt"
        "log"
        "runtime"
        "strings"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/mamba"
        "github.com/openfluke/welvet/nativerocm"
        "github.com/openfluke/welvet/quant"
)

func main() {
        if e := run(); e != nil {
                log.Fatal(e)
        }
}

func run() error {
        selected := flag.String("dtype", "all", "native parameter dtype name, or all
34")
        flag.Parse()
        if !rocmPlatformOK() {
                fmt.Println("SKIP: AMD ROCm needs linux/amd64 or windows/amd64 and
cgo")
                return nil
        }
        count := 0
        for _, dt := range core.AllDTypes {
                if *selected != "all" && !strings.EqualFold(*selected, dt.String()) {
                        continue
                }
                if e := forward(dt); e != nil {
                        if count == 0 && *selected == "all" {
                                fmt.Printf("SKIP: ROCm unavailable (%v)\n", e)
                                return nil
                        }
                        return fmt.Errorf("%s: %w", dt, e)
                }
                count++
        }
        if count == 0 {
                return fmt.Errorf("unknown dtype %q", *selected)
        }
        fmt.Printf("Mamba dtypes=%d; activations=float32; parameters=FormatNone
native dtype\n", count)
        return nil
}

func forward(dt core.DType) error {
        const d, inner = 8, 16
        in, out := make([]float32, 2*inner*d), make([]float32, d*inner)
        for i := range in {
                in[i] = float32(i%17-8) * .01
        }
        for i := range out {
                out[i] = float32(i%13-6) * .01
        }
        model, e := mamba.NewConfigured(mamba.Config{DModel: d, DState: 4, Expand: 2,
SeqLen: 3}, dt, quant.FormatNone, in, out, nil, nil)
```

```
        if e != nil {
                return e
        }
        s, e := nativerocm.Mount(model)
        if e != nil {
                return e
        }
        defer s.Close()
        x := core.NewTensor[float32](1, 3, d)
        for i := range x.Data {
                x.Data[i] = float32(i%7-3) * .05
        }
        post, e := s.Forward(x)
        if e != nil {
                return e
        }
        var checksum float64
        for _, v := range post.Data {
                checksum += float64(v)
        }
        fmt.Printf("%-10s shape=%v checksum=%.8g\n", dt, post.Shape, checksum)
        return nil
}

func rocmPlatformOK() bool {
        return runtime.GOARCH == "amd64" && (runtime.GOOS == "linux" || runtime.GOOS
== "windows")
}
```

Output

exit 0 · last run via `go run .`

```
SKIP: AMD ROCm needs linux/amd64 or windows/amd64 and cgo
```

Validate (harness)

```
cd welvet/examples/97-amd-mamba-dtypes && source ../env.sh && go run .
```

CHAPTER 98

AMD ROCm — training modes

github.com/openfluke/welvet/nativerocm `linux/windows amd64`

Why it exists

Keep repeated training updates in the mounted ROCm session.

What it is

Train two Dense layers using one of the 29 numbered modes, with persistent Step Line state and real Grid layouts for Mesh modes.

Training families

`-mode all` exercises all 29 modes. Step modes retain their Line session across calls. Mesh modes use a three-cell Grid with disabled ingress and two active layers, advancing four ticks per call. Bundled **welvet.rocm 0.79** covers train modes through StepTweenSplitSparse. These small demonstrations validate usage; they are not performance measurements.

```
go run . -mode NormalBP -steps 8
go run . -mode all -steps 8
```

Explicit mounted sessions keep updated weights native until `SyncToGo()`. Close discards any updates not returned to Go. See chapter 99 before saving.

Requirements: chapter 96.

Go example

examples/98-amd-training/main.go

Run: `cd welvet/examples/98-amd-training && source ../env.sh && go run .`

```
// Native ROCm MSE training: persistent Stack Line state or an actual Grid clock.
package main

import (
        "flag"
        "fmt"
        "log"
        "runtime"

        "github.com/openfluke/welvet/architecture"
        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/dense"
        "github.com/openfluke/welvet/layers/parallel"
        "github.com/openfluke/welvet/nativerocm"
        "github.com/openfluke/welvet/quant"
)

func main() {
        if e := run(); e != nil {
                log.Fatal(e)
        }
}

func run() error {
        selected := flag.String("mode", "NormalBP", "training mode name, or all 29")
        steps := flag.Int("steps", 8, "training calls; retain the session for Step
pipeline state")
        flag.Parse()
        if *steps < 1 {
                return fmt.Errorf("steps must be positive")
        }
        if !rocmPlatformOK() {
                fmt.Println("SKIP: AMD ROCm needs linux/amd64 or windows/amd64 and
cgo")
                return nil
        }
        if *selected == "all" {
                for mode := parallel.ModeNormalBP; mode <=
parallel.ModeStepTweenSplitSparse; mode++ {
                        if e := train(mode, *steps); e != nil {
                                if mode == parallel.ModeNormalBP {
                                        fmt.Printf("SKIP: ROCm unavailable (%v)\n",
e)
                                        return nil
                                }
                                return e
                        }
                }
                return nil
        }
        mode, e := parallel.ParseTrainMode(*selected)
        if e != nil {
                return e
        }
        if mode < parallel.ModeNormalBP || mode > parallel.ModeStepTweenSplitSparse {
                return fmt.Errorf("choose one of the 29 update modes")
        }
        if e := train(mode, *steps); e != nil {
                fmt.Printf("SKIP: ROCm unavailable (%v)\n", e)
```

```
                return nil
        }
        return nil
}

func train(mode parallel.TrainMode, steps int) error {
        var ops []any
        for layer := 0; layer < 2; layer++ {
                w := make([]float32, 16)
                for i := range w {
                        w[i] = float32(i%5-2) * .04
                }
                l, e := dense.NewConfigured(4, 4, core.ActivationTanh,
core.DTypeFloat32, quant.FormatNone, w)
                if e != nil {
                        return e
                }
                ops = append(ops, l)
        }
        stack, e := parallel.NewStack(ops...)
        if e != nil {
                return e
        }
        var model any = stack
        schedule := "Stack"
        if mode.RequiresGrid() {
                g := architecture.NewGrid(3, 1, 1, 1)
                g.Cells[0].Layer.IsDisabled = true
                for i, op := range ops {
                        g.Cells[i+1].Layer = op.(*dense.Layer).Core
                        g.Cells[i+1].Op = op
                }
                model = g
                schedule = "Grid mesh clock (4 ticks/call)"
        } else if mode.IsLineStep() {
                schedule = "Stack Line pipeline"
        }
        s, e := nativerocm.Mount(model)
        if e != nil {
                return e
        }
        defer s.Close()
        x := core.NewTensor[float32](2, 4)
        for i := range x.Data {
                x.Data[i] = float32(i%5-2) * .1
        }
        target := core.NewTensor[float32](2, 4)
        var loss float64
        for i := 0; i < steps; i++ {
                loss, e = s.TrainMSE(x, target, mode, .01, 4)
                if e != nil {
                        return fmt.Errorf("%s: %w", mode, e)
                }
        }
        if e = s.SyncToGo(); e != nil {
                return e
        }
        fmt.Printf("mode=%s schedule=%s calls=%d loss=%.8g weights=synced-to-Go\n",
mode, schedule, steps, loss)
        return nil
```

```
}

func rocmPlatformOK() bool {
        return runtime.GOARCH == "amd64" && (runtime.GOOS == "linux" || runtime.GOOS
== "windows")
}
```

Output

exit 0 · last run via `go run .`

```
SKIP: AMD ROCm needs linux/amd64 or windows/amd64 and cgo
```

Validate (harness)

```
cd welvet/examples/98-amd-training && source ../env.sh && go run .
```

CHAPTER 99

AMD ROCm — weight return and ENTITY save

github.com/openfluke/welvet/nativerocm

linux/windows amd64

Why it exists

Return trained weights to Go before executing or saving the Go model.

What it is

Train natively, explicitly SyncToGo, save an ENTITY, reload it and compare the native parameter snapshots.

Weight handoff

```
session, err := nativerocm.Mount(layer)
// Check err; defer session.Close().
// Repeat session.TrainMSE(...) as needed.
err = session.SyncToGo()
// Check err, then save through Go ENTITY IO.
```

Mount copies model state into a native session. The original Go parameters stay unchanged while that explicit session trains. `SyncToGo()` returns parameter bytes, scales and Float64 biases for Go saving. This saves model parameters, not a portable checkpoint of live Line, proxy, KV or rule state.

Use `-out /path/model.entity` to choose the output location. The default uses a temporary directory and prints it.

Requirements: chapter 96.

Go example

examples/99-amd-save-weights/main.go

Run: `cd welvet/examples/99-amd-save-weights && source ../env.sh && go run .`

```
// Explicit mount ownership, weight return to Go, ENTITY save and reload.
package main

import (
	"flag"
	"fmt"
	"log"
	"os"
	"path/filepath"
	"reflect"
	"runtime"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/dense"
	"github.com/openfluke/welvet/layers/parallel"
	"github.com/openfluke/welvet/model/entity"
	"github.com/openfluke/welvet/nativerocm"
	"github.com/openfluke/welvet/quant"
)

func main() {
	if e := run(); e != nil {
		log.Fatal(e)
	}
}

func run() error {
	out := flag.String("out", "", "ENTITY destination; default is a new temporary
directory")
	flag.Parse()
	if !rocmPlatformOK() {
		fmt.Println("SKIP: AMD ROCm needs linux/amd64 or windows/amd64 and
cgo")
		return nil
	}
	l, e := dense.NewConfigured(2, 2, core.ActivationLinear, core.DTypeFloat32,
quant.FormatNone, []float32{.3, -.2, .1, .4})
	if e != nil {
		return e
	}
	model, e := parallel.NewStack(l)
	if e != nil {
		return e
	}
	before, e := l.Weights.FlattenF32()
	if e != nil {
		return e
	}
	s, e := nativerocm.Mount(model)
	if e != nil {
		fmt.Printf("SKIP: ROCm unavailable (%v)\n", e)
		return nil
	}
	defer s.Close()
	x := &core.Tensor[float32]{Shape: []int{1, 2}, Data: []float32{.5, .25}}
	target := core.NewTensor[float32](1, 2)
```

```
        for i := 0; i < 5; i++ {
                if _, e = s.TrainMSE(x, target, parallel.ModeNormalBP, .1); e != nil
{
                        return e
                }
        }
        unchanged, e := l.Weights.FlattenF32()
        if e != nil {
                return e
        }
        if !reflect.DeepEqual(before, unchanged) {
                return fmt.Errorf("explicit mount unexpectedly changed Go parameters
before SyncToGo")
        }
        fmt.Println("Go parameters unchanged before SyncToGo")
        if e = s.SyncToGo(); e != nil {
                return e
        }
        after, e := l.Weights.FlattenF32()
        if e != nil {
                return e
        }
        if reflect.DeepEqual(before, after) {
                return fmt.Errorf("expected trained parameters after SyncToGo")
        }
        if *out == "" {
                dir, e := os.MkdirTemp("", "welvet-rocm-entity-")
                if e != nil {
                        return e
                }
                *out = filepath.Join(dir, "trained.entity")
        }
        if e = entity.WriteCameralFile(*out, model, parallel.ModeNormalBP); e != nil
{
                return e
        }
        loaded, mode, e := entity.LoadCameral(*out)
        if e != nil {
                return e
        }
        if mode != parallel.ModeNormalBP {
                return fmt.Errorf("training mode changed during save")
        }
        restored, e := nativerocm.Mount(loaded)
        if e != nil {
                return e
        }
        defer restored.Close()
        want, e := s.Weights()
        if e != nil {
                return e
        }
        got, e := restored.Weights()
        if e != nil {
                return e
        }
        if !reflect.DeepEqual(want, got) {
                return fmt.Errorf("saved parameters changed on reload")
        }
        fmt.Printf("weights returned to Go; saved=%s; reload=exact\n", *out)
```

```
        return nil
}

func rocmPlatformOK() bool {
        return runtime.GOARCH == "amd64" && (runtime.GOOS == "linux" || runtime.GOOS
== "windows")
}
```

Output

exit 0 · last run via `go run .`

```
SKIP: AMD ROCm needs linux/amd64 or windows/amd64 and cgo
```

Validate (harness)

```
cd welvet/examples/99-amd-save-weights && source ../env.sh && go run .
```

CHAPTER 100

AMD ROCm — JSON backend placement

github.com/openfluke/welvet/nativerocm `linux amd64 routing`

Why it exists

Choose which layers run on AMD GPU while retaining ordinary Go training entry points.

What it is

Enable the ROCm router and apply JSON placement to a Stack: layer 0 on Go CPU, layers 1–2 on ROCm.

Explicit placement

The example embeds placement.json; use `-config` to load another file. `nativerocm.EnableRouting()` installs the router for its owner lifetime (**Linux amd64**). `placement.Stack` applies immediate-child numeric layer keys.

Routed execution maintains native sessions and returns updated parameters to Go automatically. Changing source weights remounts a session and resets transient native state. Automatic hardware/performance planning remains separate from explicit placement.

Windows amd64 supports explicit `Mount` (chapters 96–99); automatic JSON routing is Linux-only in this release.

Requirements: chapter 96.

Go example

examples/100-amd-json-routing/main.go

Run: `cd welvet/examples/100-amd-json-routing && source ../env.sh && go run .`

```
// JSON selects backends; the existing Go Stack scheduler moves
activations/gradients.
package main

import (
        _ "embed"
        "flag"
        "fmt"
        "log"
        "os"
        "runtime"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/dense"
        "github.com/openfluke/welvet/layers/parallel"
        "github.com/openfluke/welvet/nativecpp"
        "github.com/openfluke/welvet/nativerocm"
        "github.com/openfluke/welvet/quant"
        "github.com/openfluke/welvet/runtime/placement"
)

//go:embed placement.json
var defaultPlan []byte

func main() {
        if e := run(); e != nil {
                log.Fatal(e)
        }
}

func run() error {
        path := flag.String("config", "", "optional replacement execution JSON")
        flag.Parse()
        if runtime.GOOS != "linux" || runtime.GOARCH != "amd64" {
                fmt.Println("SKIP: ROCm JSON routing needs Linux amd64 and cgo")
                return nil
        }
        raw := defaultPlan
        if *path != "" {
                var e error
                raw, e = os.ReadFile(*path)
                if e != nil {
                        return e
                }
        }
        cfg, e := nativecpp.ParseExecConfigJSON(raw)
        if e != nil {
                return e
        }
        router, e := nativerocm.EnableRouting()
        if e != nil {
                fmt.Printf("SKIP: ROCm routing unavailable (%v)\n", e)
                return nil
        }
        defer router.Close()
        var ops []any
        for i := 0; i < 3; i++ {
                l, e := dense.NewConfigured(2, 2, core.ActivationTanh,
core.DTypeFloat32, quant.FormatNone, []float32{.8, .1, -.1, .9})
```

```
                if e != nil {
                        return e
                }
                ops = append(ops, l)
        }
        model, e := parallel.NewStack(ops...)
        if e != nil {
                return e
        }
        if e = placement.Stack(model, cfg, "main", "0,0,0,0"); e != nil {
                return e
        }
        for i, op := range ops {
                fmt.Printf("layer=%d backend=%s\n", i, op.
(*dense.Layer).Exec.Backend)
        }
        x := &core.Tensor[float32]{Shape: []int{1, 2}, Data: []float32{.5, -.3}}
        target := core.NewTensor[float32](1, 2)
        loss, e := parallel.TrainStackMSE(model, x, target, parallel.ModeNormalBP,
.03)
        if e != nil {
                fmt.Printf("SKIP: ROCm train unavailable (%v)\n", e)
                return nil
        }
        stats := router.Stats()
        if stats.Backwards == 0 || stats.Updates == 0 {
                return fmt.Errorf("plan did not execute native ROCm backward/SGD")
        }
        fmt.Printf("loss=%.8g rocm routing=%+v\nRouted updates are already in Go
stores; Close releases cached mounts.\n", loss, stats)
        return nil
}
```

Output

exit 0 · last run via `go run .`

```
SKIP: ROCm JSON routing needs Linux amd64 and cgo
```

Validate (harness)

```
cd welvet/examples/100-amd-json-routing && source ../env.sh && go run .
```

CHAPTER 101

AMD ROCm — mixed Go/C++/ROCm

github.com/openfluke/welvet/nativerocm

linux amd64 routing

Why it exists

Split a single model between the independent native engines and Go.

What it is

Run a 13-layer Dense Stack: Go layers 0–3, C++ layers 4–8 and ROCm layers 9–12; verify actual native calls during training.

Three engines in one Go model

```
Go CPU [0–3] → welvet.cpp [4–8] → welvet.rocm [9–12] → Go result
```

The embedded placement.json uses `go_cpu`, `native_cpu` and `rocm`. Install both native libraries and use the normal cgo build (do **not** pass `-tags welvet_rocm`). The example reports successful C++ C ABI calls and ROCm routing counters after five NormalBP updates.

Activations and parameters cross backend boundaries. Live backend-specific Line, proxy, KV or rule state is not migrated between engines.

Requirements: Linux amd64 + chapter 96.

Go example

examples/101-amd-mixed-backends/main.go

Run: `cd welvet/examples/101-amd-mixed-backends && source ../env.sh && go run .`

```
// Keep a normal cgo build: -tags welvet_rocm intentionally disables C++.
package main

import (
	_ "embed"
	"fmt"
	"log"
	"runtime"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/dense"
	"github.com/openfluke/welvet/layers/parallel"
	"github.com/openfluke/welvet/nativecpp"
	"github.com/openfluke/welvet/nativerocm"
	"github.com/openfluke/welvet/quant"
	"github.com/openfluke/welvet/runtime/placement"
)

//go:embed placement.json
var plan []byte

func main() {
	if e := run(); e != nil {
		log.Fatal(e)
	}
}

func run() error {
	if runtime.GOOS != "linux" || runtime.GOARCH != "amd64" {
		fmt.Println("SKIP: mixed ROCm routing needs Linux amd64 and cgo")
		return nil
	}
	if !nativecpp.NativeCPUAvailable() {
		fmt.Println("SKIP: mixed routing needs the C++ artifact and normal
build without -tags welvet_rocm")
		return nil
	}
	router, e := nativerocm.EnableRouting()
	if e != nil {
		fmt.Printf("SKIP: ROCm routing unavailable (%v)\n", e)
		return nil
	}
	defer router.Close()
	cfg, e := nativecpp.ParseExecConfigJSON(plan)
	if e != nil {
		return e
	}
	var ops []any
	for i := 0; i < 13; i++ {
		l, e := dense.NewConfigured(2, 2, core.ActivationTanh,
core.DTypeFloat32, quant.FormatNone, []float32{.8, .12, -.15, .9})
		if e != nil {
			return e
		}
		ops = append(ops, l)
	}
	model, e := parallel.NewStack(ops...)
	if e != nil {
		return e
```

```
        }
        if e = placement.Stack(model, cfg, "main", "0,0,0,0"); e != nil {
                return e
        }
        x := &core.Tensor[float32]{Shape: []int{1, 2}, Data: []float32{.5, -.3}}
        target := core.NewTensor[float32](1, 2)
        before := nativecpp.CompletedCalls()
        var loss float64
        for i := 0; i < 5; i++ {
                loss, e = parallel.TrainStackMSE(model, x, target,
parallel.ModeNormalBP, .03)
                if e != nil {
                        fmt.Printf("SKIP: mixed train unavailable (%v)\n", e)
                        return nil
                }
        }
        calls := nativecpp.CompletedCalls() - before
        stats := router.Stats()
        if calls == 0 || stats.Backwards == 0 || stats.Updates == 0 {
                return fmt.Errorf("expected both C++ and ROCm native calls")
        }
        fmt.Println("Go layers 0-3 -> C++ layers 4-8 -> ROCm layers 9-12")
        fmt.Printf("loss=%.8g cpp calls=%d rocm routing=%+v\n", loss, calls, stats)
        fmt.Println("Float32 activations travel forward; gradients travel back;
routed ROCm weights return automatically to Go.")
        return nil
}
```

Output

exit 0 · last run via `go run .`

```
SKIP: mixed ROCm routing needs Linux amd64 and cgo
```

Validate (harness)

```
cd welvet/examples/101-amd-mixed-backends && source ../env.sh && go run .
```

CHAPTER 102

AMD ROCm — results and suite pointers

host-only pointers

Why it exists

Find the AMD sibling repo and live ROCm parity/timing matrices after the small usage chapters.

What it is

Print Go v1.5.0 / welvet.rocm 0.79 pins and commands for w2a go_v_rocm and rocm_smoke. No GPU required.

Where live results live

This chapter is host-only. It does not claim timings from loading the page. Sibling AMD engine: github.com/openfluke/welvet.rocm. Run the matrices on a machine with a ROCm-capable AMD GPU:

```
cd ../../w2a && go test ./tests/go_v_rocm -count=1 -timeout 180m
cd ../../welvet.rocm/examples/rocm_smoke && go run .
cd welvet/examples && ./run_amd.sh
```

Engine pin: **welvet.rocm 0.79**, model ABI **1**, Go binding `nativerocm` (Welvet **v1.5.0**). Platforms: Linux x86_64 + Windows amd64 for Mount; Linux amd64 for EnableRouting. **GPU compute untested** until a ROCm-capable HSA device is available.

Requirements overview: chapter 96.

Go example

examples/102-amd-rocm-results/main.go

Run: `cd welvet/examples/102-amd-rocm-results && source ../env.sh && go run .`

```
// Pointers to ROCm validation matrices; no GPU run is implied.
package main

import "fmt"

func main() {
        fmt.Println("AMD ROCm results pointers (not a live benchmark)")
        fmt.Println()
        fmt.Println("Engine pin: welvet.rocm 0.79 · model ABI 1 · Go binding:
nativerocm (Welvet v1.5.0)")
        fmt.Println("Usage tour: 96 mount · 97 Mamba dtypes · 98 train modes · 99
SyncToGo/ENTITY · 100 JSON · 101 mixed")
        fmt.Println()
        fmt.Println("Live matrices (sibling checkouts):")
        fmt.Println("  cd ../../w2a && go test ./tests/go_v_rocm -count=1 -timeout
180m")
        fmt.Println("  cd ../../welvet.rocm/examples/rocm_smoke && go run .")
        fmt.Println()
        fmt.Println("Install shared library into Go:")
        fmt.Println("  cd ../../welvet.rocm && ./scripts/linux_install_go.sh")
        fmt.Println()
        fmt.Println("Small usage tour without the large matrix:")
        fmt.Println("  ../run_amd.sh")
}
```

Output

exit 0 · last run via `go run .`

```
AMD ROCm results pointers (not a live benchmark)

Engine pin: welvet.rocm 0.79 · model ABI 1 · Go binding: nativerocm (Welvet v1.5.0)
Usage tour: 96 mount · 97 Mamba dtypes · 98 train modes · 99 SyncToGo/ENTITY · 100
JSON · 101 mixed

Live matrices (sibling checkouts):
  cd ../../w2a && go test ./tests/go_v_rocm -count=1 -timeout 180m
  cd ../../welvet.rocm/examples/rocm_smoke && go run .

Install shared library into Go:
  cd ../../welvet.rocm && ./scripts/linux_install_go.sh

Small usage tour without the large matrix:
  ../run_amd.sh
```

Validate (harness)

```
cd welvet/examples/102-amd-rocm-results && source ../env.sh && go run .
```

CHAPTER 103

Vulkan — mount and forward

github.com/openfluke/welvet/nativevulkan `linux amd64`

Why it exists

Run a Go layer on a Vulkan GPU with an explicit lifetime via the independent welvet.vulkan library.

What it is

Mount an identity Dense layer, inspect the Vulkan version and device identity, and forward a Float32 tensor.

Session ownership

Create a session with `nativevulkan.Mount`, call `Forward` repeatedly and close it when finished. Mount accepts supported layers, Stack, Parallel and Grid layouts. The library is independent of welvet.cpp, welvet.metal, welvet.cuda and welvet.rocm.

Requirements and backend boundaries

Go Welvet **v1.6.0** adds the independent **welvet.vulkan 0.79** library (mounted model ABI 1) at github.com/openfluke/welvet.vulkan. Install from its checkout:

```
cd ../welvet.vulkan
./scripts/linux_install_go.sh
```

Execution requires **Linux x86_64**, Go with cgo, and a Vulkan 1.x-capable device + loader/runtime. Set `WELVET_VULKAN_LIBRARY` to override library discovery. Ordinary Go CPU use does not require a Vulkan GPU. Unsupported hosts print SKIP; missing libraries or no GPU on a supported host also soft-SKIP in these book examples.

Honest coverage: no silent host fallback labeled as Vulkan (see sibling `docs/VULKAN_ONLY.md`). LSTM/RNN/Mamba/GDN gates still use host `exp` / `tanh` after GPU MatVec — same pattern as cuda. Windows artifacts are not shipped yet.

Go schedules work and manages memory; numerical Vulkan forward, backward and weight updates execute SPIR-V compute when a device is present. Activations are Float32; FormatNone parameter storage follows the Vulkan dtype matrix. Small examples demonstrate APIs; GPU launch costs can outweigh useful work at small shapes.

`cpu-tiled` / `go_cpu` means Go CPU; `native_cpu` means welvet.cpp; `vulkan` means the independent Vulkan GPU library. Examples 103–106 can use `go run -tags welvet_vulkan .` to exclude C++ linkage. Chapter 108 requires the normal cgo build for both native engines. JSON routing (§107–§108) is **Linux amd64**.

Go example

examples/103-vulkan-mount/main.go

Run: `cd welvet/examples/103-vulkan-mount && source ../env.sh && go run .`

```
// Mount one layer, execute its forward pass on Vulkan, and release it.
package main

import (
	"flag"
	"fmt"
	"log"
	"runtime"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/dense"
	"github.com/openfluke/welvet/nativevulkan"
	"github.com/openfluke/welvet/quant"
)

func main() {
	if e := run(); e != nil {
		log.Fatal(e)
	}
}

func run() error {
	library := flag.String("library", "", "optional libwelvet_vulkan.so/.dll
path")
	flag.Parse()
	if !vulkanPlatformOK() {
		fmt.Println("SKIP: Vulkan needs linux/amd64 and cgo")
		return nil
	}
	l, e := dense.NewConfigured(2, 2, core.ActivationLinear, core.DTypeFloat32,
quant.FormatNone, []float32{1, 0, 0, 1})
	if e != nil {
		return e
```

```
	}
	session, e := nativevulkan.Mount(l, nativevulkan.Options{LibraryPath:
*library})
	if e != nil {
		fmt.Printf("SKIP: Vulkan unavailable (%v)\n", e)
		return nil
	}
	defer session.Close()
	x := &core.Tensor[float32]{Shape: []int{1, 2}, Data: []float32{.5, -.25}}
	post, e := session.Forward(x)
	if e != nil {
		return e
	}
	fmt.Printf("backend=vulkan version=%s\ndevice=%s\ninput=%v output=%v
shape=%v\n", session.Version(), session.Identity(), x.Data, post.Data, post.Shape)
	return nil
}

func vulkanPlatformOK() bool {
	return runtime.GOOS == "linux" && runtime.GOARCH == "amd64"
}
```

Output

exit 0 · last run via `go run .`

```
SKIP: Vulkan needs linux/amd64 and cgo
```

Validate (harness)

```
cd welvet/examples/103-vulkan-mount && source ../env.sh && go run .
```

CHAPTER 104

Vulkan — Mamba parameter dtypes

github.com/openfluke/welvet/nativevulkan `linux amd64`

Why it exists

Exercise native storage formats without changing the activation interface.

What it is

Run a seeded Mamba forward pass across all 34 native parameter dtypes, or select one with -dtype.

Storage versus arithmetic

`-dtype all` sweeps native parameter representations. The Vulkan execution interface uses Float32 activations. Quantized FormatNone dtype storage and packed quantization formats are different coverage claims.

```
go run . -dtype all
go run . -dtype float16
```

Requirements and library installation: chapter 103.

Go example

examples/104-vulkan-mamba-dtypes/main.go

Run:

```
cd welvet/examples/104-vulkan-mamba-dtypes && source ../env.sh && go run .
```

```
// Float32 tensors with independently selected native parameter encodings.
package main
```

```
import (
        "flag"
        "fmt"
        "log"
        "runtime"
        "strings"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/mamba"
        "github.com/openfluke/welvet/nativevulkan"
        "github.com/openfluke/welvet/quant"
)

func main() {
        if e := run(); e != nil {
                log.Fatal(e)
        }
}

func run() error {
        selected := flag.String("dtype", "all", "native parameter dtype name, or all
34")
        flag.Parse()
        if !vulkanPlatformOK() {
                fmt.Println("SKIP: Vulkan needs linux/amd64 and cgo")
                return nil
        }
        count := 0
        for _, dt := range core.AllDTypes {
                if *selected != "all" && !strings.EqualFold(*selected, dt.String()) {
                        continue
                }
                if e := forward(dt); e != nil {
                        if count == 0 && *selected == "all" {
                                fmt.Printf("SKIP: Vulkan unavailable (%v)\n", e)
                                return nil
                        }
                        return fmt.Errorf("%s: %w", dt, e)
                }
                count++
        }
        if count == 0 {
                return fmt.Errorf("unknown dtype %q", *selected)
        }
        fmt.Printf("Mamba dtypes=%d; activations=float32; parameters=FormatNone
native dtype\n", count)
        return nil
}

func forward(dt core.DType) error {
        const d, inner = 8, 16
        in, out := make([]float32, 2*inner*d), make([]float32, d*inner)
        for i := range in {
                in[i] = float32(i%17-8) * .01
        }
        for i := range out {
                out[i] = float32(i%13-6) * .01
        }
        model, e := mamba.NewConfigured(mamba.Config{DModel: d, DState: 4, Expand: 2,
SeqLen: 3}, dt, quant.FormatNone, in, out, nil, nil)
```

```
        if e != nil {
                return e
        }
        s, e := nativevulkan.Mount(model)
        if e != nil {
                return e
        }
        defer s.Close()
        x := core.NewTensor[float32](1, 3, d)
        for i := range x.Data {
                x.Data[i] = float32(i%7-3) * .05
        }
        post, e := s.Forward(x)
        if e != nil {
                return e
        }
        var checksum float64
        for _, v := range post.Data {
                checksum += float64(v)
        }
        fmt.Printf("%-10s shape=%v checksum=%.8g\n", dt, post.Shape, checksum)
        return nil
}

func vulkanPlatformOK() bool {
        return runtime.GOOS == "linux" && runtime.GOARCH == "amd64"
}
```

Output

exit 0 · last run via `go run .`

```
SKIP: Vulkan needs linux/amd64 and cgo
```

Validate (harness)

```
cd welvet/examples/104-vulkan-mamba-dtypes && source ../env.sh && go run .
```

CHAPTER 105

Vulkan — training modes

github.com/openfluke/welvet/nativevulkan `linux amd64`

Why it exists

Keep repeated training updates in the mounted Vulkan session.

What it is

Train two Dense layers using one of the 29 numbered modes, with persistent Step Line state and real Grid layouts for Mesh modes.

Training families

`-mode all` exercises all 29 modes. Step modes retain their Line session across calls. Mesh modes use a three-cell Grid with disabled ingress and two active layers, advancing four ticks per call. Bundled **welvet.vulkan 0.79** covers train modes through StepTweenSplitSparse. These small demonstrations validate usage; they are not performance measurements.

```
go run . -mode NormalBP -steps 8
go run . -mode all -steps 8
```

Explicit mounted sessions keep updated weights native until `SyncToGo()`. Close discards any updates not returned to Go. See chapter 106 before saving.

Requirements: chapter 103.

Go example

examples/105-vulkan-training/main.go

Run: `cd welvet/examples/105-vulkan-training && source ../env.sh && go run .`

```
// Native Vulkan MSE training: persistent Stack Line state or an actual Grid clock.
package main

import (
	"flag"
	"fmt"
	"log"
	"runtime"

	"github.com/openfluke/welvet/architecture"
	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/dense"
	"github.com/openfluke/welvet/layers/parallel"
	"github.com/openfluke/welvet/nativevulkan"
	"github.com/openfluke/welvet/quant"
)

func main() {
	if e := run(); e != nil {
		log.Fatal(e)
	}
}

func run() error {
	selected := flag.String("mode", "NormalBP", "training mode name, or all 29")
	steps := flag.Int("steps", 8, "training calls; retain the session for Step
pipeline state")
	flag.Parse()
	if *steps < 1 {
		return fmt.Errorf("steps must be positive")
	}
	if !vulkanPlatformOK() {
		fmt.Println("SKIP: Vulkan needs linux/amd64 and cgo")
		return nil
	}
	if *selected == "all" {
		for mode := parallel.ModeNormalBP; mode <=
parallel.ModeStepTweenSplitSparse; mode++ {
			if e := train(mode, *steps); e != nil {
				if mode == parallel.ModeNormalBP {
					fmt.Printf("SKIP: Vulkan unavailable (%v)\n",
e)
					return nil
				}
				return e
			}
		}
		return nil
	}
	mode, e := parallel.ParseTrainMode(*selected)
	if e != nil {
		return e
	}
	if mode < parallel.ModeNormalBP || mode > parallel.ModeStepTweenSplitSparse {
		return fmt.Errorf("choose one of the 29 update modes")
	}
	if e := train(mode, *steps); e != nil {
		fmt.Printf("SKIP: Vulkan unavailable (%v)\n", e)
		return nil
```

```
        }
        return nil
}

func train(mode parallel.TrainMode, steps int) error {
        var ops []any
        for layer := 0; layer < 2; layer++ {
                w := make([]float32, 16)
                for i := range w {
                        w[i] = float32(i%5-2) * .04
                }
                l, e := dense.NewConfigured(4, 4, core.ActivationTanh,
core.DTypeFloat32, quant.FormatNone, w)
                if e != nil {
                        return e
                }
                ops = append(ops, l)
        }
        stack, e := parallel.NewStack(ops...)
        if e != nil {
                return e
        }
        var model any = stack
        schedule := "Stack"
        if mode.RequiresGrid() {
                g := architecture.NewGrid(3, 1, 1, 1)
                g.Cells[0].Layer.IsDisabled = true
                for i, op := range ops {
                        g.Cells[i+1].Layer = op.(*dense.Layer).Core
                        g.Cells[i+1].Op = op
                }
                model = g
                schedule = "Grid mesh clock (4 ticks/call)"
        } else if mode.IsLineStep() {
                schedule = "Stack Line pipeline"
        }
        s, e := nativevulkan.Mount(model)
        if e != nil {
                return e
        }
        defer s.Close()
        x := core.NewTensor[float32](2, 4)
        for i := range x.Data {
                x.Data[i] = float32(i%5-2) * .1
        }
        target := core.NewTensor[float32](2, 4)
        var loss float64
        for i := 0; i < steps; i++ {
                loss, e = s.TrainMSE(x, target, mode, .01, 4)
                if e != nil {
                        return fmt.Errorf("%s: %w", mode, e)
                }
        }
        if e = s.SyncToGo(); e != nil {
                return e
        }
        fmt.Printf("mode=%s schedule=%s calls=%d loss=%.8g weights=synced-to-Go\n",
mode, schedule, steps, loss)
        return nil
}
```

```
func vulkanPlatformOK() bool {
        return runtime.GOOS == "linux" && runtime.GOARCH == "amd64"
}
```

Output

exit 0 · last run via `go run .`

```
SKIP: Vulkan needs linux/amd64 and cgo
```

Validate (harness)

```
cd welvet/examples/105-vulkan-training && source ../env.sh && go run .
```

CHAPTER 106

Vulkan — weight return and ENTITY save

github.com/openfluke/welvet/nativevulkan `linux amd64`

Why it exists

Return trained weights to Go before executing or saving the Go model.

What it is

Train natively, explicitly SyncToGo, save an ENTITY, reload it and compare the native parameter snapshots.

Weight handoff

```
session, err := nativevulkan.Mount(layer)
// Check err; defer session.Close().
// Repeat session.TrainMSE(...) as needed.
err = session.SyncToGo()
// Check err, then save through Go ENTITY IO.
```

Mount copies model state into a native session. The original Go parameters stay unchanged while that explicit session trains. `SyncToGo()` returns parameter bytes, scales and Float64 biases for Go saving. This saves model parameters, not a portable checkpoint of live Line, proxy, KV or rule state.

Use `-out /path/model.entity` to choose the output location. The default uses a temporary directory and prints it.

Requirements: chapter 103.

Go example

examples/106-vulkan-save-weights/main.go

Run:

```
cd welvet/examples/106-vulkan-save-weights && source ../env.sh && go run .
```

```
// Explicit mount ownership, weight return to Go, ENTITY save and reload.
package main

import (
	"flag"
	"fmt"
	"log"
	"os"
	"path/filepath"
	"reflect"
	"runtime"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/dense"
	"github.com/openfluke/welvet/layers/parallel"
	"github.com/openfluke/welvet/model/entity"
	"github.com/openfluke/welvet/nativevulkan"
	"github.com/openfluke/welvet/quant"
)

func main() {
	if e := run(); e != nil {
		log.Fatal(e)
	}
}

func run() error {
	out := flag.String("out", "", "ENTITY destination; default is a new temporary
directory")
	flag.Parse()
	if !vulkanPlatformOK() {
		fmt.Println("SKIP: Vulkan needs linux/amd64 and cgo")
		return nil
	}
	l, e := dense.NewConfigured(2, 2, core.ActivationLinear, core.DTypeFloat32,
quant.FormatNone, []float32{.3, -.2, .1, .4})
	if e != nil {
		return e
	}
	model, e := parallel.NewStack(l)
	if e != nil {
		return e
	}
	before, e := l.Weights.FlattenF32()
	if e != nil {
		return e
	}
	s, e := nativevulkan.Mount(model)
	if e != nil {
		fmt.Printf("SKIP: Vulkan unavailable (%v)\n", e)
		return nil
	}
	defer s.Close()
	x := &core.Tensor[float32]{Shape: []int{1, 2}, Data: []float32{.5, .25}}
	target := core.NewTensor[float32](1, 2)
	for i := 0; i < 5; i++ {
```

```
                if _, e = s.TrainMSE(x, target, parallel.ModeNormalBP, .1); e != nil
{
                        return e
                }
        }
        unchanged, e := l.Weights.FlattenF32()
        if e != nil {
                return e
        }
        if !reflect.DeepEqual(before, unchanged) {
                return fmt.Errorf("explicit mount unexpectedly changed Go parameters
before SyncToGo")
        }
        fmt.Println("Go parameters unchanged before SyncToGo")
        if e = s.SyncToGo(); e != nil {
                return e
        }
        after, e := l.Weights.FlattenF32()
        if e != nil {
                return e
        }
        if reflect.DeepEqual(before, after) {
                return fmt.Errorf("expected trained parameters after SyncToGo")
        }
        if *out == "" {
                dir, e := os.MkdirTemp("", "welvet-vulkan-entity-")
                if e != nil {
                        return e
                }
                *out = filepath.Join(dir, "trained.entity")
        }
        if e = entity.WriteCameralFile(*out, model, parallel.ModeNormalBP); e != nil
{
                return e
        }
        loaded, mode, e := entity.LoadCameral(*out)
        if e != nil {
                return e
        }
        if mode != parallel.ModeNormalBP {
                return fmt.Errorf("training mode changed during save")
        }
        restored, e := nativevulkan.Mount(loaded)
        if e != nil {
                return e
        }
        defer restored.Close()
        want, e := s.Weights()
        if e != nil {
                return e
        }
        got, e := restored.Weights()
        if e != nil {
                return e
        }
        if !reflect.DeepEqual(want, got) {
                return fmt.Errorf("saved parameters changed on reload")
        }
        fmt.Printf("weights returned to Go; saved=%s; reload=exact\n", *out)
        return nil
```

```
}

func vulkanPlatformOK() bool {
        return runtime.GOOS == "linux" && runtime.GOARCH == "amd64"
}
```

Output

exit 0 · last run via `go run .`

```
SKIP: Vulkan needs linux/amd64 and cgo
```

Validate (harness)

```
cd welvet/examples/106-vulkan-save-weights && source ../env.sh && go run .
```

CHAPTER 107

Vulkan — JSON backend placement

github.com/openfluke/welvet/nativevulkan `linux amd64 routing`

Why it exists

Choose which layers run on Vulkan while retaining ordinary Go training entry points.

What it is

Enable the Vulkan router and apply JSON placement to a Stack: layer 0 on Go CPU, layers 1–2 on Vulkan.

Explicit placement

The example embeds placement.json; use `-config` to load another file. `nativevulkan.EnableRouting()` installs the router for its owner lifetime (**Linux amd64**). `placement.Stack` applies immediate-child numeric layer keys.

Routed execution maintains native sessions and returns updated parameters to Go automatically. Changing source weights remounts a session and resets transient native state. Automatic hardware/performance planning remains separate from explicit placement.

Requirements: chapter 103.

Go example

examples/107-vulkan-json-routing/main.go

Run:

```
cd welvet/examples/107-vulkan-json-routing && source ../env.sh && go run .
```

```
// JSON selects backends; the existing Go Stack scheduler moves
activations/gradients.
package main
```

```
import (
        _ "embed"
        "flag"
        "fmt"
        "log"
        "os"
        "runtime"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/dense"
        "github.com/openfluke/welvet/layers/parallel"
        "github.com/openfluke/welvet/nativecpp"
        "github.com/openfluke/welvet/nativevulkan"
        "github.com/openfluke/welvet/quant"
        "github.com/openfluke/welvet/runtime/placement"
)

//go:embed placement.json
var defaultPlan []byte

func main() {
        if e := run(); e != nil {
                log.Fatal(e)
        }
}

func run() error {
        path := flag.String("config", "", "optional replacement execution JSON")
        flag.Parse()
        if runtime.GOOS != "linux" || runtime.GOARCH != "amd64" {
                fmt.Println("SKIP: Vulkan JSON routing needs Linux amd64 and cgo")
                return nil
        }
        raw := defaultPlan
        if *path != "" {
                var e error
                raw, e = os.ReadFile(*path)
                if e != nil {
                        return e
                }
        }
        cfg, e := nativecpp.ParseExecConfigJSON(raw)
        if e != nil {
                return e
        }
        router, e := nativevulkan.EnableRouting()
        if e != nil {
                fmt.Printf("SKIP: Vulkan routing unavailable (%v)\n", e)
                return nil
        }
        defer router.Close()
        var ops []any
        for i := 0; i < 3; i++ {
                l, e := dense.NewConfigured(2, 2, core.ActivationTanh,
core.DTypeFloat32, quant.FormatNone, []float32{.8, .1, -.1, .9})
                if e != nil {
                        return e
                }
                ops = append(ops, l)
```

```
        }
        model, e := parallel.NewStack(ops...)
        if e != nil {
                return e
        }
        if e = placement.Stack(model, cfg, "main", "0,0,0,0"); e != nil {
                return e
        }
        for i, op := range ops {
                fmt.Printf("layer=%d backend=%s\n", i, op.
(*dense.Layer).Exec.Backend)
        }
        x := &core.Tensor[float32]{Shape: []int{1, 2}, Data: []float32{.5, -.3}}
        target := core.NewTensor[float32](1, 2)
        loss, e := parallel.TrainStackMSE(model, x, target, parallel.ModeNormalBP,
.03)
        if e != nil {
                fmt.Printf("SKIP: Vulkan train unavailable (%v)\n", e)
                return nil
        }
        stats := router.Stats()
        if stats.Backwards == 0 || stats.Updates == 0 {
                return fmt.Errorf("plan did not execute native Vulkan backward/SGD")
        }
        fmt.Printf("loss=%.8g vulkan routing=%+v\nRouted updates are already in Go
stores; Close releases cached mounts.\n", loss, stats)
        return nil
}
```

Output

exit 0 · last run via `go run .`

```
SKIP: Vulkan JSON routing needs Linux amd64 and cgo
```

Validate (harness)

```
cd welvet/examples/107-vulkan-json-routing && source ../env.sh && go run .
```

CHAPTER 108

Vulkan — mixed Go/C++/Vulkan

github.com/openfluke/welvet/nativevulkan `linux amd64 routing`

Why it exists

Split a single model between the independent native engines and Go.

What it is

Run a 13-layer Dense Stack: Go layers 0–3, C++ layers 4–8 and Vulkan layers 9–12; verify actual native calls during training.

Three engines in one Go model

```
Go CPU [0–3] → welvet.cpp [4–8] → welvet.vulkan [9–12] → Go result
```

The embedded placement.json uses `go_cpu`, `native_cpu` and `vulkan`. Install both native libraries and use the normal cgo build (do **not** pass `-tags welvet_vulkan`). The example reports successful C++ C ABI calls and Vulkan routing counters after five NormalBP updates.

Activations and parameters cross backend boundaries. Live backend-specific Line, proxy, KV or rule state is not migrated between engines.

Requirements: Linux amd64 + chapter 103.

Go example

examples/108-vulkan-mixed-backends/main.go

Run:

```
cd welvet/examples/108-vulkan-mixed-backends && source ../env.sh && go run .
```

```
// Keep a normal cgo build: -tags welvet_vulkan intentionally disables C++.
package main

import (
	_ "embed"
	"fmt"
	"log"
	"runtime"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/dense"
	"github.com/openfluke/welvet/layers/parallel"
	"github.com/openfluke/welvet/nativecpp"
	"github.com/openfluke/welvet/nativevulkan"
	"github.com/openfluke/welvet/quant"
	"github.com/openfluke/welvet/runtime/placement"
)

//go:embed placement.json
var plan []byte

func main() {
	if e := run(); e != nil {
		log.Fatal(e)
	}
}

func run() error {
	if runtime.GOOS != "linux" || runtime.GOARCH != "amd64" {
		fmt.Println("SKIP: mixed Vulkan routing needs Linux amd64 and cgo")
		return nil
	}
	if !nativecpp.NativeCPUAvailable() {
		fmt.Println("SKIP: mixed routing needs the C++ artifact and normal
build without -tags welvet_vulkan")
		return nil
	}
	router, e := nativevulkan.EnableRouting()
	if e != nil {
		fmt.Printf("SKIP: Vulkan routing unavailable (%v)\n", e)
		return nil
	}
	defer router.Close()
	cfg, e := nativecpp.ParseExecConfigJSON(plan)
	if e != nil {
		return e
	}
	var ops []any
	for i := 0; i < 13; i++ {
		l, e := dense.NewConfigured(2, 2, core.ActivationTanh,
core.DTypeFloat32, quant.FormatNone, []float32{.8, .12, -.15, .9})
		if e != nil {
			return e
		}
		ops = append(ops, l)
	}
	model, e := parallel.NewStack(ops...)
	if e != nil {
		return e
```

```
        }
        if e = placement.Stack(model, cfg, "main", "0,0,0,0"); e != nil {
                return e
        }
        x := &core.Tensor[float32]{Shape: []int{1, 2}, Data: []float32{.5, -.3}}
        target := core.NewTensor[float32](1, 2)
        before := nativecpp.CompletedCalls()
        var loss float64
        for i := 0; i < 5; i++ {
                loss, e = parallel.TrainStackMSE(model, x, target,
parallel.ModeNormalBP, .03)
                if e != nil {
                        fmt.Printf("SKIP: mixed train unavailable (%v)\n", e)
                        return nil
                }
        }
        calls := nativecpp.CompletedCalls() - before
        stats := router.Stats()
        if calls == 0 || stats.Backwards == 0 || stats.Updates == 0 {
                return fmt.Errorf("expected both C++ and Vulkan native calls")
        }
        fmt.Println("Go layers 0-3 -> C++ layers 4-8 -> Vulkan layers 9-12")
        fmt.Printf("loss=%.8g cpp calls=%d vulkan routing=%+v\n", loss, calls, stats)
        fmt.Println("Float32 activations travel forward; gradients travel back;
routed Vulkan weights return automatically to Go.")
        return nil
}
```

Output

exit 0 · last run via `go run .`

```
SKIP: mixed Vulkan routing needs Linux amd64 and cgo
```

Validate (harness)

```
cd welvet/examples/108-vulkan-mixed-backends && source ../env.sh && go run .
```

CHAPTER 109

Vulkan — results and suite pointers

host-only pointers

Why it exists

Find the Vulkan sibling repo and live go_v_vulkan matrices after the small usage chapters.

What it is

Print Go v1.6.0 / welvet.vulkan 0.79 pins and commands for w2a go_v_vulkan. No GPU required.

Where live results live

This chapter is host-only. It does not claim timings from loading the page. Sibling Vulkan engine: github.com/openfluke/welvet.vulkan. Run the matrices on a machine with a Vulkan-capable GPU:

```
cd ../../w2a && go test ./tests/go_v_vulkan -count=1 -timeout 180m
cd ../../welvet.vulkan && ./scripts/build_linux.sh && ./build/example_vulkan_smoke
cd welvet/examples && ./run_vulkan.sh
```

Engine pin: **welvet.vulkan 0.79**, model ABI **1**, Go binding `nativevulkan` (Welvet **v1.6.0**). Platform: Linux x86_64 for Mount and EnableRouting. Pure-Vulkan contract: no silent CPU path labeled Vulkan.

Requirements overview: chapter 103.

Go example

examples/109-vulkan-results/main.go

Run: `cd welvet/examples/109-vulkan-results && source ../env.sh && go run .`

```
// Pointers to Vulkan validation matrices; no GPU run is implied.
package main

import "fmt"

func main() {
        fmt.Println("Vulkan results pointers (not a live benchmark)")
        fmt.Println()
        fmt.Println("Engine pin: welvet.vulkan 0.79 · model ABI 1 · Go binding:
nativevulkan (Welvet v1.6.0)")
        fmt.Println("Usage tour: 103 mount · 104 Mamba dtypes · 105 train modes · 106
SyncToGo/ENTITY · 107 JSON · 108 mixed")
        fmt.Println()
        fmt.Println("Live matrices (sibling checkouts):")
        fmt.Println("  cd ../../w2a && go test ./tests/go_v_vulkan -count=1 -timeout
180m")
        fmt.Println("  cd ../../welvet.vulkan && ./scripts/build_linux.sh &&
./build/example_vulkan_smoke")
        fmt.Println()
        fmt.Println("Install shared library into Go:")
        fmt.Println("  cd ../../welvet.vulkan && ./scripts/linux_install_go.sh")
        fmt.Println()
        fmt.Println("Small usage tour without the large matrix:")
        fmt.Println("  ../run_vulkan.sh")
}
```

Output

exit 0 · last run via `go run .`

```
Vulkan results pointers (not a live benchmark)

Engine pin: welvet.vulkan 0.79 · model ABI 1 · Go binding: nativevulkan (Welvet
v1.6.0)
Usage tour: 103 mount · 104 Mamba dtypes · 105 train modes · 106 SyncToGo/ENTITY ·
107 JSON · 108 mixed

Live matrices (sibling checkouts):
  cd ../../w2a && go test ./tests/go_v_vulkan -count=1 -timeout 180m
  cd ../../welvet.vulkan && ./scripts/build_linux.sh && ./build/example_vulkan_smoke

Install shared library into Go:
  cd ../../welvet.vulkan && ./scripts/linux_install_go.sh

Small usage tour without the large matrix:
  ../run_vulkan.sh
```

Validate (harness)

```
cd welvet/examples/109-vulkan-results && source ../env.sh && go run .
```

CHAPTER 110

Apple MLX — SyncToMLX and forward

github.com/openfluke/welvet/nativemlx `darwin arm64`

Why it exists

Run a Go Dense layer on Apple MLX GPU with an explicit SyncToMLX lifetime — forward-only, no Mount/TrainMSE twin.

What it is

Install welvet.mlx, SyncToMLX an identity Dense, set BackendMLX, and forward a Float32 tensor.

User-owned sync

MLX does not use `nativemetal.Mount`. Weights stay in Go Stores until you call `SyncToMLX` (caps-check + capture). Then set `Exec.Backend = core.BackendMLX` (or use `core.PreferMLX()` when the dylib + Metal GPU are ready) and call the ordinary layer `Forward`.

```
must(layer.SyncToMLX())
layer.Exec.Backend = core.BackendMLX
_, post, err := dense.Forward(layer, x)
```

Requirements and honesty

Go Welvet **v1.7.0** adds the independent **welvet.mlx 1.0.0** library at github.com/openfluke/welvet.mlx. Install into the Go tree:

```
cd ../welvet.mlx
./scripts/macos_install_go.sh
```

Needs **Apple silicon macOS**, cgo, shipped `libwelvet_mlx.dylib`, and a Metal GPU. `nativemlx.Available()` is false without Metal — there is **no CPU fallback labeled as MLX**

and no silent fallthrough to `welvet.metal`. Other hosts print SKIP.

Forward-only: no TrainMSE / NormalBP / SyncToGo train loop on MLX. `SyncFromMLX` exists for API symmetry (mirror → Go) but forward does not mutate W. Caps reject unsupported layer×dtype mounts.

Sibling contract: docs/MLX_ONLY.md. Caps tour: §111. Dense + Sequential: §112. Live matrices: §113. Metal train twin remains §82–88.

Go example

examples/110-apple-mlx/main.go

Run: `cd welvet/examples/110-apple-mlx && source ../env.sh && go run .`

```
// SyncToMLX one Dense layer, forward on Apple MLX GPU, release nothing explicit
// (shared DefaultSession lives for the process).
package main

import (
	"fmt"
	"log"
	"runtime"

	"github.com/openfluke/welvet/core"
	"github.com/openfluke/welvet/layers/dense"
	"github.com/openfluke/welvet/nativemlx"
	"github.com/openfluke/welvet/quant"
)

func main() {
	if err := run(); err != nil {
		log.Fatal(err)
	}
}

func run() error {
	if runtime.GOOS != "darwin" || runtime.GOARCH != "arm64" {
		fmt.Println("SKIP: Apple MLX needs Apple silicon macOS and cgo")
		return nil
	}
	fmt.Println("version", nativemlx.Version())
	fmt.Println("available", nativemlx.Available())
	fmt.Println("prefer", core.PreferMLX())
	if !nativemlx.Available() {
		fmt.Println("SKIP: need libwelvet_mlx.dylib + Metal GPU
(../welvet.mlx/scripts/macos_install_go.sh)")
		return nil
	}

	l, err := dense.NewConfigured(2, 2, core.ActivationLinear, core.DTypeFloat32,
quant.FormatNone, []float32{1, 0, 0, 1})
```

```
        if err != nil {
                return err
        }
        if err := l.SyncToMLX(); err != nil {
                return err
        }
        l.Exec.Backend = core.BackendMLX

        x := &core.Tensor[float32]{Shape: []int{1, 2}, Data: []float32{0.5, -0.25}}
        _, post, err := dense.Forward(l, x)
        if err != nil {
                return err
        }
        fmt.Printf("backend=mlx synced=%v\ninput=%v output=%v shape=%v\n",
l.MLXSynced(), x.Data, post.Data, post.Shape)
        return nil
}
```

Output

exit 0 · last run via `go run .`

```
version 1.0.0
available true
prefer mlx
backend=mlx synced=true
input=[0.5 -0.25] output=[0.5 -0.25] shape=[1 2]
```

Validate (harness)

```
cd welvet/examples/110-apple-mlx && source ../env.sh && go run .
```

CHAPTER 111

Apple MLX — caps and dtype rejects

github.com/openfluke/welvet/nativemlx `darwin arm64`

Why it exists

Unsupported mounts must hard-fail before any GPU work — not soft-fallback to Go labeled as MLX.

What it is

Print SupportedLayers, accept dense/float32, and show RequireLayer rejecting dense/float64.

Caps table

`nativemlx.LayerSupported(layer, dtype)` and `SupportedLayers()` mirror the welvet.mlx C ABI caps. `SyncToMLX` / `RequireLayer` refuse anything outside that table. Dense GPU compute rejects **float64**; host I/O stays Float32 activations with FormatNone parameter storage.

welvet.mlx **1.0.0** caps list the signed leaf ops (Dense, Softmax, RMSNorm, LayerNorm, CNN2/3, ConvT2/3, LSTM, Embedding, Residual, SwiGLU, MHA). **Sequential** is a container that mounts supported children — see §112. Full dtype matrix: FORWARD_CHECKLIST.md.

Requirements: chapter 110.

Go example

examples/111-apple-mlx-caps/main.go

Run: `cd welvet/examples/111-apple-mlx-caps && source ../env.sh && go run .`

```
// Print MLX caps and demonstrate RequireLayer rejects.
package main
```

```
import (
        "fmt"
        "log"
        "runtime"
        "strings"

        "github.com/openfluke/welvet/nativemlx"
)

func main() {
        if err := run(); err != nil {
                log.Fatal(err)
        }
}

func run() error {
        if runtime.GOOS != "darwin" || runtime.GOARCH != "arm64" {
                fmt.Println("SKIP: Apple MLX needs Apple silicon macOS and cgo")
                return nil
        }
        if !nativemlx.Available() {
                fmt.Println("SKIP: need libwelvet_mlx.dylib + Metal GPU")
                return nil
        }

        layers := nativemlx.SupportedLayers()
        fmt.Println("version", nativemlx.Version())
        fmt.Println("supported_layers", strings.Join(layers, ","))
        fmt.Println("dense/float32", nativemlx.LayerSupported("dense", "float32"))
        fmt.Println("dense/float64", nativemlx.LayerSupported("dense", "float64"))

        if err := nativemlx.RequireLayer("dense", "float32"); err != nil {
                return fmt.Errorf("expected dense/float32 ok: %w", err)
        }
        fmt.Println("RequireLayer dense/float32: ok")

        err := nativemlx.RequireLayer("dense", "float64")
        if err == nil {
                return fmt.Errorf("expected dense/float64 reject")
        }
        fmt.Println("RequireLayer dense/float64:", err)
        return nil
}
```

Output

exit 0 · last run via `go run .`

```
version 1.0.0
supported_layers
dense,softmax,rmsnorm,layernorm,cnn2,cnn3,convt2,convt3,lstm,embedding,residual,swigl
u,mha
dense/float32 true
dense/float64 false
RequireLayer dense/float32: ok
```

```
RequireLayer dense/float64: nativemlx: reject dense/float64 — not supported on MLX
GPU (see SupportedLayers)
```

Validate (harness)

```
cd welvet/examples/111-apple-mlx-caps && source ../env.sh && go run .
```

CHAPTER 112

Apple MLX — Dense prepare and Sequential

github.com/openfluke/welvet/nativemlx `darwin arm64`

Why it exists

Show prepare-once Dense execute and a small Sequential tower on the same MLX session.

What it is

SyncToMLX a Dense with PrepareMLX, then SyncToMLX a two-Dense Sequential and forward once.

Prepare vs online

Dense can `PrepareMLX(batch)` outside the timed loop so repeated `Forward` calls stay execute-only. Sequential rebuilds a welvet.mlx Sequential from supported children (Dense / Softmax / RMSNorm / LayerNorm today) and host-ping-pongs child MLX evals.

These shapes are API demos — not performance claims. Live large-shape matrices live in w2a `go_v_mlx` (§113).

Requirements: chapter 110.

Go example

examples/112-apple-mlx-forward/main.go

Run: `cd welvet/examples/112-apple-mlx-forward && source ../env.sh && go run .`

```
// Dense PrepareMLX + Sequential SyncToMLX forward on Apple MLX.
package main

import (
        "fmt"
        "log"
```

```
        "runtime"

        "github.com/openfluke/welvet/core"
        "github.com/openfluke/welvet/layers/dense"
        "github.com/openfluke/welvet/layers/sequential"
        "github.com/openfluke/welvet/nativemlx"
        "github.com/openfluke/welvet/quant"
)

func main() {
        if err := run(); err != nil {
                log.Fatal(err)
        }
}

func run() error {
        if runtime.GOOS != "darwin" || runtime.GOARCH != "arm64" {
                fmt.Println("SKIP: Apple MLX needs Apple silicon macOS and cgo")
                return nil
        }
        if !nativemlx.Available() {
                fmt.Println("SKIP: need libwelvet_mlx.dylib + Metal GPU")
                return nil
        }

        if err := demoDensePrepare(); err != nil {
                return err
        }
        return demoSequential()
}

func demoDensePrepare() error {
        init := make([]float32, 4*4)
        for i := range init {
                init[i] = float32(i+1) * 0.05
        }
        l, err := dense.NewConfigured(4, 4, core.ActivationLinear, core.DTypeFloat32,
quant.FormatNone, init)
        if err != nil {
                return err
        }
        if err := l.SyncToMLX(); err != nil {
                return err
        }
        if err := l.PrepareMLX(1); err != nil {
                return err
        }
        l.Exec.Backend = core.BackendMLX
        x := core.NewTensor[float32](1, 4)
        for i := range x.Data {
                x.Data[i] = 0.1 * float32(i+1)
        }
        _, post, err := dense.Forward(l, x)
        if err != nil {
                return err
        }
        fmt.Printf("dense prepare+exec out=%v\n", post.Data)
        return nil
}
```

```
func demoSequential() error {
        // Two identity Dense children (Depth=2, Dim=3).
        packed := append(identity3(), identity3()...)
        seq, err := sequential.NewConfigured(sequential.Config{Dim: 3, Depth: 2},
core.DTypeFloat32, quant.FormatNone, packed)
        if err != nil {
                return err
        }
        if err := seq.SyncToMLX(); err != nil {
                return err
        }
        seq.Exec.Backend = core.BackendMLX
        x := &core.Tensor[float32]{Shape: []int{1, 3}, Data: []float32{1, 2, 3}}
        _, post, err := sequential.Forward(seq, x)
        if err != nil {
                return err
        }
        fmt.Printf("sequential mlx out=%v synced=%v\n", post.Data, seq.MLXSynced())
        return nil
}

func identity3() []float32 {
        return []float32{
                1, 0, 0,
                0, 1, 0,
                0, 0, 1,
        }
}
```

Output

exit 0 · last run via `go run .`

```
dense prepare+exec out=[0.15 0.35 0.5500001 0.75]
sequential mlx out=[1 2 3] synced=true
```

Validate (harness)

```
cd welvet/examples/112-apple-mlx-forward && source ../env.sh && go run .
```

CHAPTER 113

Apple MLX — go_v_mlx results pointers

github.com/openfluke/w2a/suites/go_v_mlx

host-only pointers

Why it exists

Find the sibling harness and live MLX parity matrices after the small usage chapters.

What it is

Print Go v1.7.0 / welvet.mlx 1.0.0 pins and commands for w2a go_v_mlx. No GPU required.

Where live results live

This chapter is host-only. It does not claim timings from loading the page. Sibling MLX engine: github.com/openfluke/welvet.mlx. Suite README: w2a/suites/go_v_mlx.

```
cd ../../w2a && go test ./tests/go_v_mlx -count=1 -timeout 60m
cd ../../welvet.mlx && ./scripts/macos_install_go.sh
cd welvet/examples && ./run_mlx.sh
```

Engine pin: **welvet.mlx 1.0.0**, Go binding `nativemlx` (Welvet **v1.7.0**). Platform: Apple silicon macOS for SyncToMLX / BackendMLX. Pure-MLX contract: no silent CPU path labeled MLX; no TrainMSE on MLX.

Requirements overview: chapter 110.

Go example

examples/113-apple-mlx-results/main.go

Run: `cd welvet/examples/113-apple-mlx-results && source ../env.sh && go run .`

```
// Pointers to MLX validation matrices; no GPU run is implied.
package main

import "fmt"

func main() {
        fmt.Println("Apple MLX results pointers (not a live benchmark)")
        fmt.Println()
        fmt.Println("Engine pin: welvet.mlx 1.0.0 · Go binding: nativemlx (Welvet
v1.7.0)")
        fmt.Println("Usage tour: 110 SyncToMLX · 111 caps · 112 Dense/Sequential ·
113 pointers")
        fmt.Println()
        fmt.Println("Live matrices (sibling checkouts):")
        fmt.Println("  cd ../../w2a && go test ./tests/go_v_mlx -count=1 -timeout
60m")
        fmt.Println("  cd ../../welvet.mlx && ./scripts/macos_build_w2a.sh")
        fmt.Println()
        fmt.Println("Install shared library into Go:")
        fmt.Println("  cd ../../welvet.mlx && ./scripts/macos_install_go.sh")
        fmt.Println()
        fmt.Println("Small usage tour without the large matrix:")
        fmt.Println("  ../run_mlx.sh")
}
```

Output

exit 0 · last run via `go run .`

```
Apple MLX results pointers (not a live benchmark)

Engine pin: welvet.mlx 1.0.0 · Go binding: nativemlx (Welvet v1.7.0)
Usage tour: 110 SyncToMLX · 111 caps · 112 Dense/Sequential · 113 pointers

Live matrices (sibling checkouts):
  cd ../../w2a && go test ./tests/go_v_mlx -count=1 -timeout 60m
  cd ../../welvet.mlx && ./scripts/macos_build_w2a.sh

Install shared library into Go:
  cd ../../welvet.mlx && ./scripts/macos_install_go.sh

Small usage tour without the large matrix:
  ../run_mlx.sh
```

Validate (harness)

```
cd welvet/examples/113-apple-mlx-results && source ../env.sh && go run .
```