Scaling NBN, rounds 3 and 4: does a larger budget buy a better answer?

Threads, GPU, memory and time; conditioning, anytime certified sampling, bounded conditioning

Alfredo (`alfredo_claude`) for Giovanni Briglia

17 October 2026 — branch `feat/scaling-mechanisms`, local, not pushed

Summary

Question. If NBN gets more workers (threads, a GPU), more memory or more time, does it give a better answer? **Round 3** measured this on the existing methods. **Round 4** adds three opt-in options so that exact and certified inference can also use a larger budget. Defaults and semantics are unchanged.

- **Before (round 3).** Sampling, fitting and amortized methods use a larger budget: a 4× budget moves the largest graph finished by 2–10×, and LW error falls like $1/\sqrt{\text{particles}}$. Exact VE does not: its limit is the elimination width, and 4× memory adds only $\log_3 4 \approx 1.26$ to it ($16 \to 17$). Time could not help VE at all.
- **Conditioning (round 4, change 1).** Exact answers when VE does not fit. All 40 conditioned answers that have a VE reference match it to $1.7 \cdot 10^{-7}$. At 1 GB on the degree-3 graphs, plain VE answers 6/10 queries (0/10 on MPS); conditioning answers 8–10/10. Workers now divide the time of an exact answer: for the same query, 20 CPU threads are about 5× faster than 1 thread, and the TITAN X 30–100×.
- **Anytime certified sampling (change 2).** Stop on time or on a target width with a valid 95% interval; the width falls like $1/\sqrt{t}$, and every interval covered the exact answer (584/584 runs where the exact answer is known). Collapsed sampling (`collapse="auto"`) returns the exact answer when the query fits, and helps on the GPU; on CPU it is slower than plain sampling on dense graphs.
- **Bounded conditioning (change 3).** Sound bounds with no probability, exact at the end. In practice they stay near $[0, 1]$ on these networks: with evidence, the unseen prior mass dwarfs $P(e)$. An honest negative result; a tighter bound is sketched in §4.
- **Hard limit.** At average degree ≥ 4.5 the cutsets have 10–24 variables (3^{10}–3^{24} instantiations): exact answers would need 10^3–10^7 s even on the GPU. There, only sampling answers, and the certified interval is the honest output.

1 The question and what theory predicts

A budget has three parts: *workers* (CPU threads, a GPU), *memory* and *time*. A method "scales with the budget" if a larger budget gives a faster answer, a larger model, or a better answer. The experts (`koller_claude`: Koller & Friedman 2009; `moss_claude`: Kochenderfer et al., *Algorithms for Validation*; `badings_claude`: interval MDPs) agreed on what each method can do:

Table 1: What each budget part can buy, by method (theory; n variables, K values, width w).

method	workers	memory	time
exact VE	faster contractions (large factors only)	width grows only by $\log_K$ of the memory factor: VE needs $O(nK^{w+1})$ (K&F §9.4)	nothing, before round 4: VE is all or nothing
conditioning on a cutset C (round 4)	instantiations are independent, so they run in parallel	a smaller C fits a larger budget	time $\approx nK^{\lvert C\rvert+w_C+1}$, space $\approx nK^{w_C+1}$ (K&F §9.5, Thm 9.11–9.13): time buys width
sampling (LW, certified)	particles are independent	larger batches	error and certified width shrink like $1/\sqrt{t}$; anytime-valid stopping is needed to stop on the data (Howard et al. 2021)
bounded conditioning (round 4)	as conditioning	as conditioning	sound bounds that tighten as more instantiations are enumerated (K&F §12.5; Horvitz et al. 1989)
fitting	batched (stacked) fits	larger batches	more epochs; portfolios cost the sum of the candidates

Two points from the experts shaped round 4. **(i)** Before round 4, time did not help exact inference at all: a query either fitted the memory budget or was refused. Conditioning (K&F §9.5, pp. 315–321) trades time for memory with no loss of exactness, and is never cheaper than VE when VE fits (Thm 9.13), so it must be a fallback, not a default. **(ii)** Sampling already turns time into accuracy, but stopping "when the interval is narrow enough" breaks a fixed-n certificate: *Algorithms for Validation* gives only fixed-m bounds (Ex. 7.3, p. 196), so an anytime-valid scheme is needed. For interval MDPs, the `badings_claude` expert noted that confidence should be spent only on data intervals, while solving should keep sound deterministic bounds and keep the best bounds seen so far (Haddad & Monmege 2018); round 4 follows the same rule.

2 Round 3: does NBN use a larger budget?

Measured on the Apple M5 (4 performance + 6 efficiency cores, 32 GB, MPS) before any round-4 change. The Mac was shared with another session's jobs during part of the runs: those rows were re-run with an idle gate, and the remaining timings are conservative.

2.1 Threads

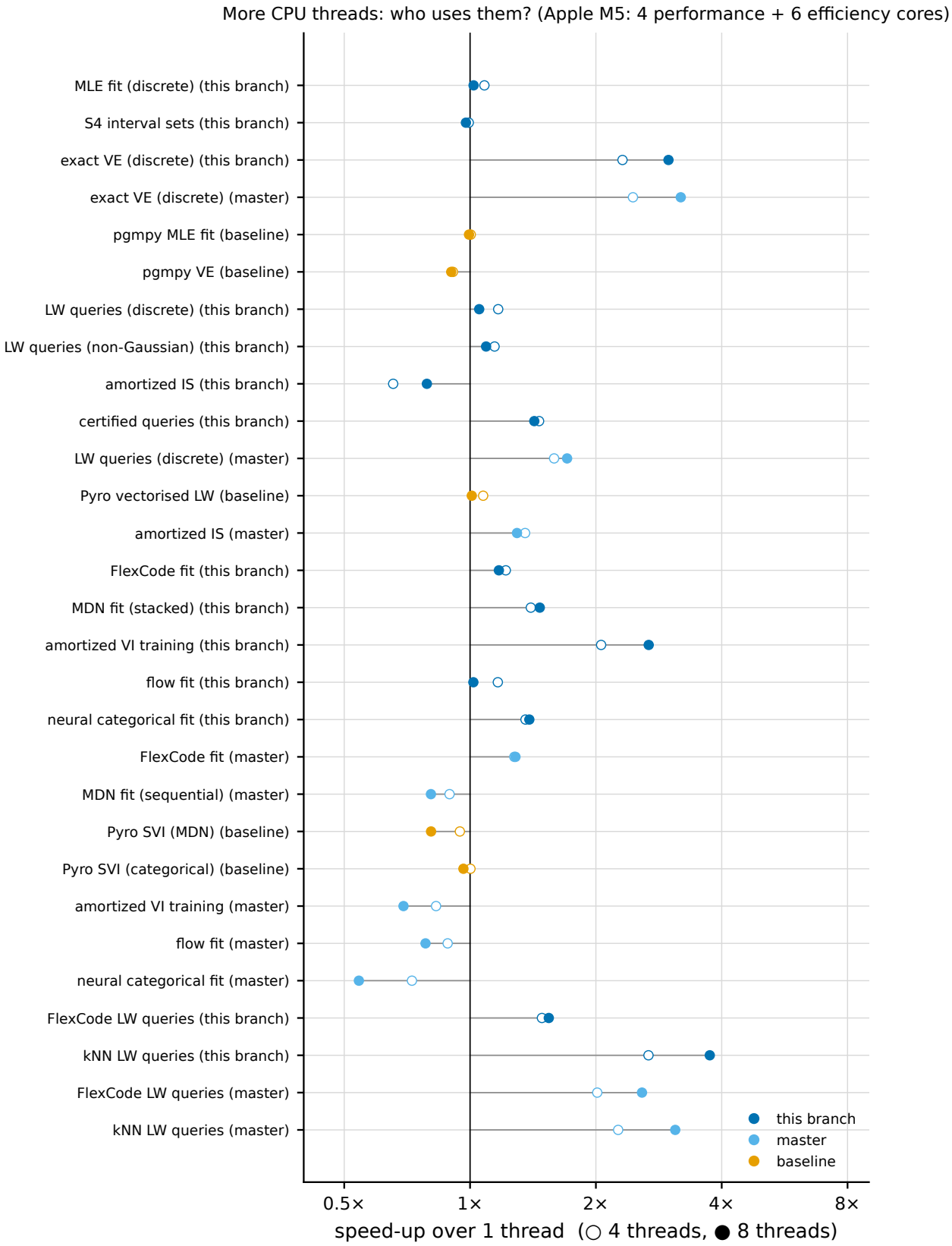


Figure 1: Speed-up from 1 to 8 CPU threads.

Table 2: R3-1. CPU threads: time (s) at 1, 2, 4, 8 threads and speed-up 1→8 (budget 120 s, 16 GB; same size per row)

computation	version	size	1	2	4	8	speed-up 1→4	speed-up 1→8
MLE fit (discrete)	this branch	5000	1.51	1.48	1.4	1.49	1.08×	1.02×
MLE fit (discrete)	master	5000	error	error	error	error	–	–
pgmpy MLE fit	baseline	5000	3.97	4.14	3.95	3.99	1.00×	0.99×
exact VE (discrete)	this branch	3	10.5	6.82	4.55	3.53	2.32×	2.98×
exact VE (discrete)	master	3	10.5	6.23	4.29	3.29	2.45×	3.19×
pgmpy VE	baseline	2	0.909	1.02	0.999	1.01	0.91×	0.90×
S4 interval sets	this branch	8	1.83	1.89	1.85	1.87	0.99×	0.98×
MDN fit (sequential)	master	200	19.2	20.4	21.5	23.8	0.89×	0.81×
MDN fit (stacked)	this branch	500	15.1	11.8	10.8	10.3	1.40×	1.47×
Pyro SVI (MDN)	baseline	100	13.3	14.2	14.1	16.5	0.95×	0.81×
amortized VI training	this branch	500	14	8.89	6.82	5.25	2.06×	2.68×
amortized VI training	master	500	24.9	25.6	30.1	36	0.83×	0.69×
FlexCode fit	master	200	13.5	11.1	10.6	10.5	1.28×	1.28×
FlexCode fit	this branch	1000	18.5	16.2	15.2	15.8	1.22×	1.17×
flow fit	master	100	18.3	19.4	20.7	23.3	0.88×	0.78×
flow fit	this branch	200	15.4	13.8	13.2	15.2	1.17×	1.02×
neural categorical fit	master	500	10.8	12.9	14.9	20	0.73×	0.54×
neural categorical fit	this branch	1000	10	7.78	7.37	7.21	1.36×	1.39×
Pyro SVI (categorical)	baseline	500	23.9	24.1	23.8	24.8	1.00×	0.96×
FlexCode LW queries	this branch	1000	2.43	1.88	1.64	1.58	1.49×	1.54×
FlexCode LW queries	master	50	17.4	10.9	8.62	6.74	2.01×	2.58×
kNN LW queries	this branch	100	13.9	7.92	5.22	3.72	2.67×	3.75×
kNN LW queries	master	10	21.4	12	9.48	6.92	2.26×	3.10×
LW queries (non-Gaussian)	this branch	5000	1.85	1.67	1.61	1.69	1.14×	1.09×
Pyro vectorised LW	baseline	1000	11.8	11.5	11	11.7	1.07×	1.01×
LW queries (discrete)	this branch	1000	11.3	9.63	9.67	10.7	1.17×	1.05×
LW queries (discrete)	master	1000	84.8	58.6	53.3	49.7	1.59×	1.71×
certified queries	this branch	500	15.7	12.1	10.7	11	1.46×	1.42×
amortized IS	this branch	1000	1.2	1.53	1.83	1.52	0.65×	0.79×
amortized IS	master	1000	9.78	7.7	7.22	7.55	1.35×	1.30×

Threads help where the work is a few large tensor operations: exact VE (3.0×), kNN likelihood weighting (3.75×), amortized VI training (2.7×; master was slower with more threads, 0.69×). They do little for MLE fitting, S4 sets and most samplers, whose cost is a Python loop over nodes.

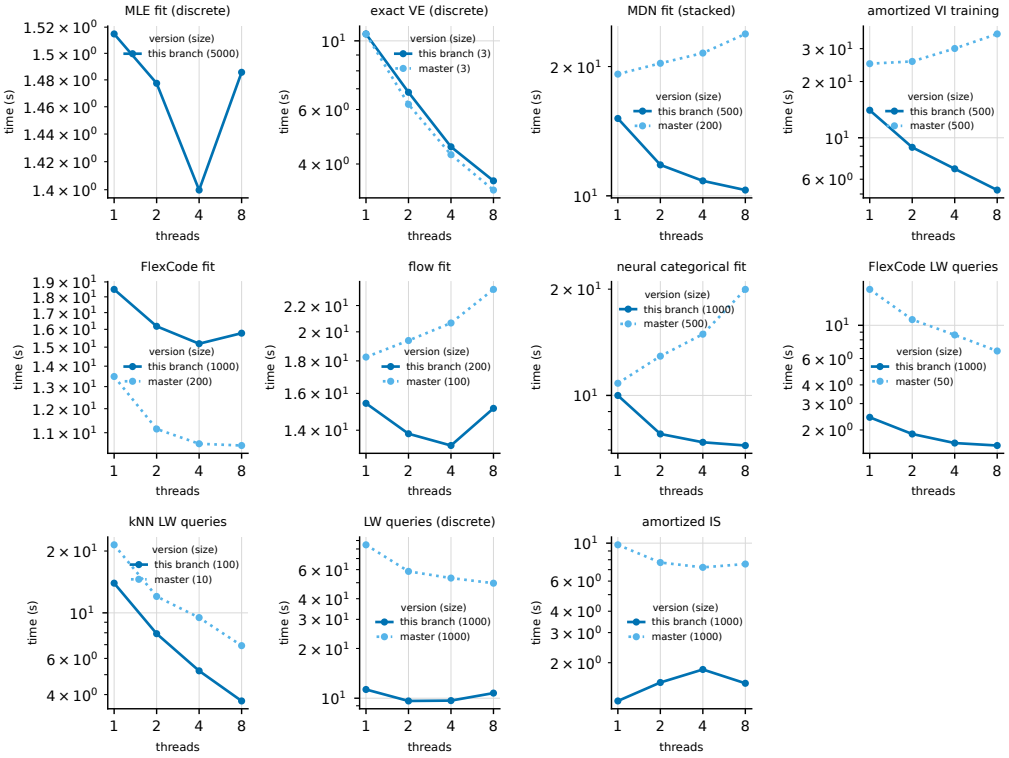


Figure 2: Threads: this branch vs master at the same sizes.

2.2 GPU

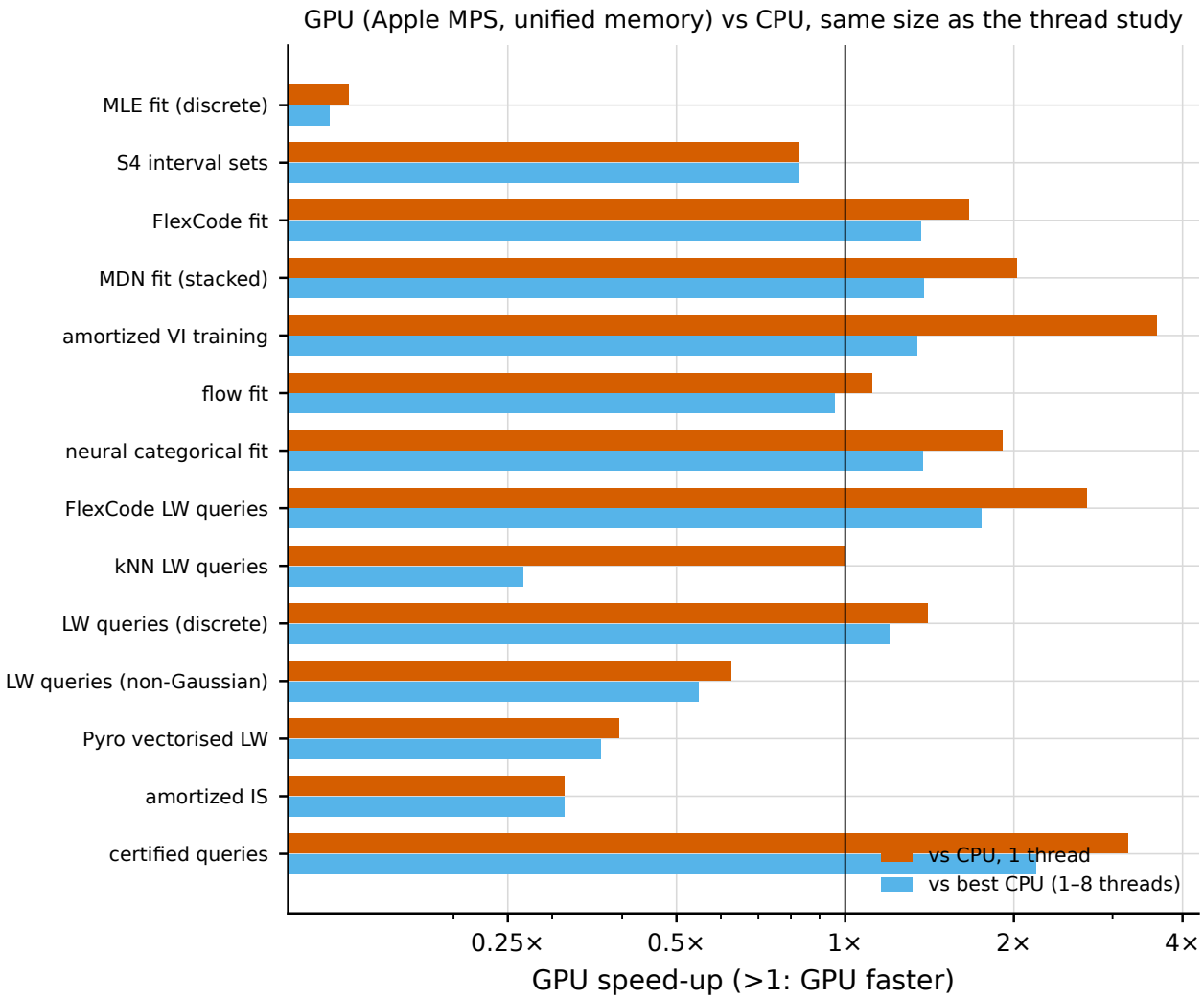


Figure 3: GPU (Apple MPS) vs CPU.

Table 3: R3-2. GPU (Apple MPS) vs CPU: time (s); speed-up = CPU time / GPU time. * exact VE: the guard answers a different number of queries per device, so times are not comparable

computation	size	CPU 1 thread	CPU best (threads)	GPU	GPU vs 1 thread	GPU vs best CPU
MLE fit (discrete)	5000	1.51	1.4 (4)	11.6	0.13×	0.12×
S4 interval sets	8	1.83	1.83 (1)	2.21	0.83×	0.83×
exact VE (discrete) *	3	10.5 (9/10)	3.53 (8)	6.41 (6/10)	–	–
FlexCode fit	1000	18.5	15.2 (4)	11.1	1.66×	1.37×
MDN fit (stacked)	500	15.1	10.3 (8)	7.46	2.03×	1.38×
Pyro SVI (MDN)	100	13.3	13.3 (1)	timeout	–	–
Pyro SVI (categorical)	500	23.9	23.8 (4)	timeout	–	–
amortized VI training	500	14	5.25 (8)	3.9	3.60×	1.35×
flow fit	200	15.4	13.2 (4)	13.8	1.12×	0.96×
neural categorical fit	1000	10	7.21 (8)	5.24	1.91×	1.38×
FlexCode LW queries	1000	2.43	1.58 (8)	0.9	2.70×	1.75×
kNN LW queries	100	13.9	3.72 (8)	14	1.00×	0.27×
LW queries (discrete)	1000	11.3	9.63 (2)	8.03	1.41×	1.20×
LW queries (non-Gaussian)	5000	1.85	1.61 (4)	2.94	0.63×	0.55×
Pyro vectorised LW	1000	11.8	11 (4)	30	0.39×	0.37×
amortized IS	1000	1.2	1.2 (1)	3.79	0.32×	0.32×
certified queries	500	15.7	10.7 (4)	4.9	3.20×	2.19×

The GPU helps batched neural work (amortized VI 3.6×, certified queries 3.2×, FlexCode LW 2.7×) and hurts small or loop-bound work (MLE fit 0.13×, amortized IS 0.32×). Exact VE on MPS answered fewer

queries: MPS tensors have at most 16 dimensions, and its allocator does not reuse freed blocks within a query, so the memory guard is stricter. Round 4 removes the first limit.

2.3 Frontier: the largest graph finished

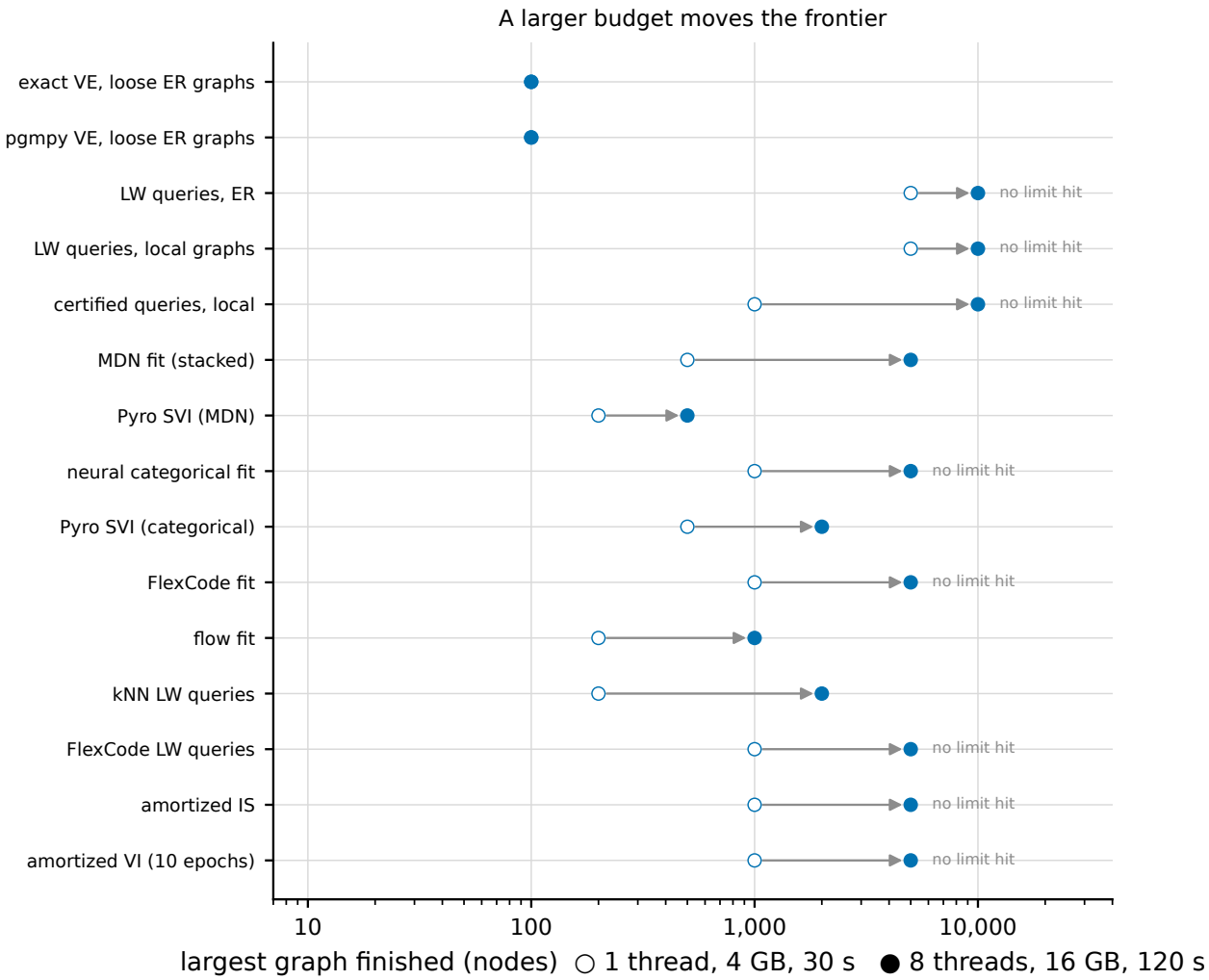


Figure 4: Largest graph finished, reference budget (1 thread, 4 GB, 30 s) vs large budget (8 threads, 16 GB, 120 s).

Table 4: R3-3. Largest graph (nodes) finished: reference budget (1 thread, 4 GB, 30 s) vs large budget (8 threads, 16 GB, 120 s)

computation	reference: largest	stopped by	large: largest	stopped by	gain in nodes
exact VE, loose ER graphs	100	ok at 200	100	ok at 200	1×
pgmpy VE, loose ER graphs	100	oom at 200	100	crash at 200	1×
LW queries, ER	5e+03	none (max tested)	1e+04	none (max tested)	2×
LW queries, local graphs	5e+03	none (max tested)	1e+04	none (max tested)	2×
certified queries, local	1e+03	timeout at 2000	1e+04	none (max tested)	10×
MDN fit (stacked)	500	timeout at 1000	5e+03	timeout at 10000	10×
Pyro SVI (MDN)	200	timeout at 500	500	timeout at 1000	2×
neural categorical fit	1e+03	none (max tested)	5e+03	none (max tested)	5×
Pyro SVI (categorical)	500	timeout at 1000	2e+03	timeout at 5000	4×
FlexCode fit	1e+03	none (max tested)	5e+03	none (max tested)	5×
flow fit	200	timeout at 500	1e+03	timeout at 2000	5×
kNN LW queries	200	timeout at 500	2e+03	timeout at 5000	10×
FlexCode LW queries	1e+03	none (max tested)	5e+03	none (max tested)	5×
amortized IS	1e+03	none (max tested)	5e+03	none (max tested)	5×
amortized VI (10 epochs)	1e+03	none (max tested)	5e+03	none (max tested)	5×

A 4× budget moves the frontier by 2–10× in nodes for every method *except* exact VE, whose limit is the elimination width, not the number of nodes.

2.4 Exact VE: width, not size

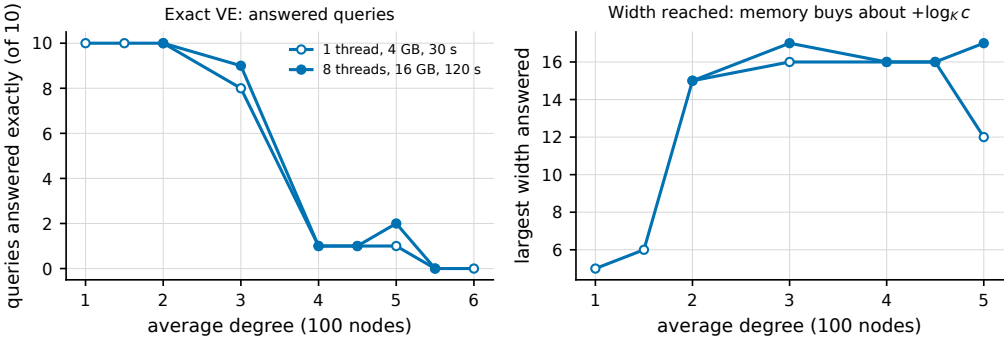


Figure 5: Exact VE on denser graphs (100 nodes, $K = 3$).

Table 5: R3-4. Exact inference on denser graphs (100 nodes, 10 queries): queries answered and the largest elimination width answered

avg degree	budget	NBN answered	NBN max width answered	NBN time (s)	pgmpy
1	1 thread, 4 GB, 30 s	10/10	5	0.0108	10/10 (0.00925 s)
1.5	1 thread, 4 GB, 30 s	10/10	6	0.0132	10/10 (0.0111 s)
2	1 thread, 4 GB, 30 s	10/10	15	0.268	10/10 (0.927 s)
3	1 thread, 4 GB, 30 s	8/10	16	5.03	timeout
4	1 thread, 4 GB, 30 s	1/10	16	2.53	–
4.5	1 thread, 4 GB, 30 s	1/10	16	2.8	oom
5	1 thread, 4 GB, 30 s	1/10	12	0.379	–
5.5	1 thread, 4 GB, 30 s	0/10	–	oom	–
6	1 thread, 4 GB, 30 s	0/10	–	oom	–
2	8 threads, 16 GB, 120 s	10/10	15	0.145	10/10 (1.01 s)
3	8 threads, 16 GB, 120 s	9/10	17	3.51	oom
4	8 threads, 16 GB, 120 s	1/10	16	0.875	–
4.5	8 threads, 16 GB, 120 s	1/10	16	1.01	–
5	8 threads, 16 GB, 120 s	2/10	17	1.8	–
5.5	8 threads, 16 GB, 120 s	0/10	–	oom	–

With 4× the memory the largest width answered goes from 16 to 17, as predicted: $\log_3 4 \approx 1.26$. Memory alone cannot rescue dense graphs. This is the motivation for conditioning (round 4).

2.5 Particles

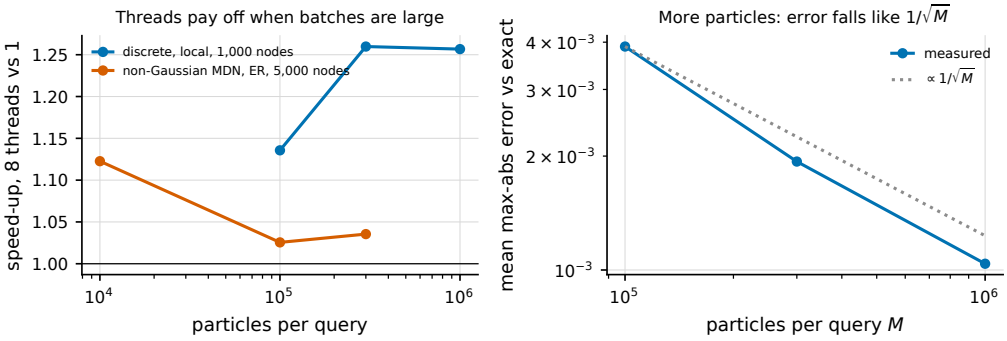


Figure 6: Likelihood weighting with more particles.

Table 6: R3-5. Likelihood weighting with more particles: time (s) for 10 queries at 1 and 8 threads, and mean max-abs error vs exact VE (discrete only)

network	particles	1 thread	8 threads	speed-up	error vs exact	ESS/N
discrete, local, 1,000 nodes	100,000	12.1	10.6	1.14×	0.0039	0.314
discrete, local, 1,000 nodes	300,000	36.1	28.7	1.26×	0.0019	0.316
discrete, local, 1,000 nodes	1,000,000	126	100	1.26×	0.0010	0.315
non-Gaussian MDN, ER, 5,000 nodes	10,000	1.83	1.63	1.12×	–	0.164
non-Gaussian MDN, ER, 5,000 nodes	100,000	16.6	16.2	1.03×	–	0.167
non-Gaussian MDN, ER, 5,000 nodes	300,000	51.6	49.8	1.04×	–	0.167

Time buys accuracy as theory says: error 0.0039 → 0.0019 → 0.0010 for $10^5 \to 3 \cdot 10^5 \to 10^6$ particles ($\propto 1/\sqrt{M}$). Threads give only 1.26× even at 10^6 particles: the sampler loops over nodes in topological order, and each step is too small to split well. Running *queries* in parallel would use the workers better (next steps).

3 Round 4: three opt-in options that turn budget into answers

Nothing changes by default: every option below must be asked for. The default `certified_query` path gives bit-identical results (test), and plain VE still raises `OutOfMemoryError` when a query does not fit.

Table 7: Round-4 options (commits `f6d19fe`, `3a5ff28`, `870d2ab`).

option	what it does	guarantee
`TensorVariableElimination( on_oom="condition"); HybridRouter( fallback="condition"); conditioned_query(...)`	when VE does not fit memory (or, on MPS, needs more than 16 tensor dimensions), clamp a greedy cutset C (Bidyuk & Dechter 2007) and sum the exact VE answers over its instantiations, many at once as rows of one batched elimination	exact (equal to VE to 10^{-7} in tests down to 1% of the memory VE needs); refuses beyond `max_conditioning` instantiations, and the router then falls back to LW
`certified_query(..., time_budget_s=, target_width=)`	anytime certified sampling: checkpoints $n_j = n_0 2^{j-1}$ at level $\delta_j = \delta/(j(j+1))$; the answer is the intersection of all checkpoint intervals	all checkpoint intervals hold together with probability $\geq 1-\delta$ (union bound; the simplest "stitching" of Howard et al. 2021), whatever the stopping rule; never wider with more time
`certified_query(..., collapse="auto"|[vars])`	collapsed (cutset) sampling: sample only the ancestral closure of a cutset, compute the rest exactly (K&F §12.4)	same certificate (per-particle statistics stay in $[0, W]$, K&F Eq. 12.11); exact when nothing is left to sample
`conditioned_query(..., time_budget_s= or enumerate_limit=)`	bounded conditioning: enumerate instantiations in order of prior frequency; unseen mass U bounds the rest: $[A_y/(A+U),\ (A_y+U)/(A+U)]$ (K&F §12.5, Eqs. 12.35–36)	sound with probability 1 (deterministic); exact when all instantiations are enumerated

Devices. All options run on CPU, CUDA and MPS. On MPS the planner respects the 16-dimension and 2^{27}-element tensor limits and keeps float64 accumulators on the CPU. Tests: the full unit suite passes on the Mac (CPU and MPS); the inference tests pass on the server (NVIDIA TITAN X Pascal 12 GB, torch 2.7.1+cu126; only MPS-specific tests skip).

Machines. CPU and CUDA results come from the server (20 cores, 125 GB, TITAN X); MPS results from the Mac (idle-gated). E1: 100-node ER graphs, $K = 3$, 10 queries with 5 evidence variables, average degree 3, 4.5 and 6, memory budgets 1, 4, 16 GB (10 GB on CUDA); at most 10^6 instantiations and 20 s per query (a slower query is stopped and its time projected from its measured rate). E2: 4 GB, time budgets 0.25–16 s. E3: 1 GB, 1–64 s.

3.1 E1. Conditioning: time and workers now buy exact answers

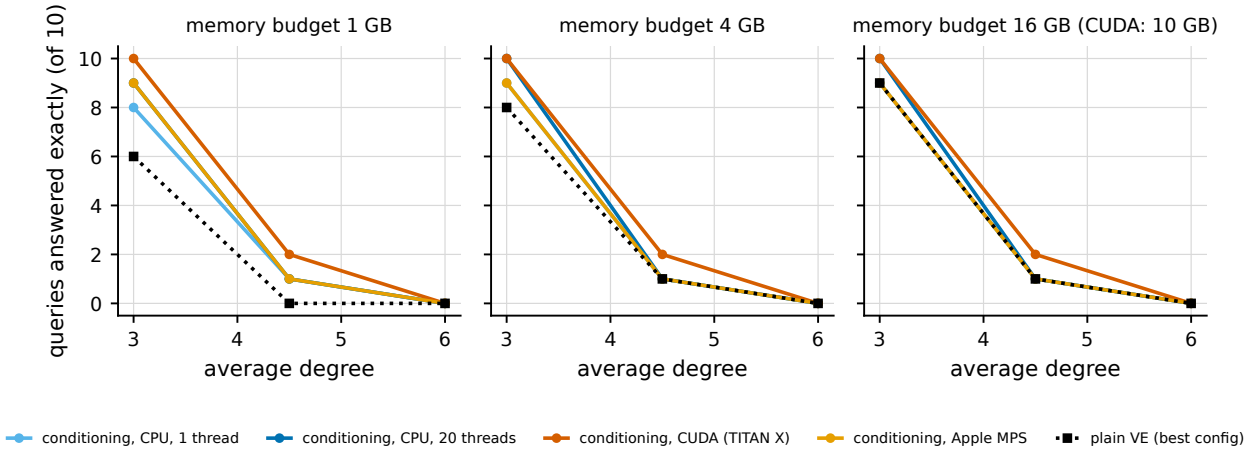


Figure 7: Queries answered exactly within 20 s, by memory budget and graph density.

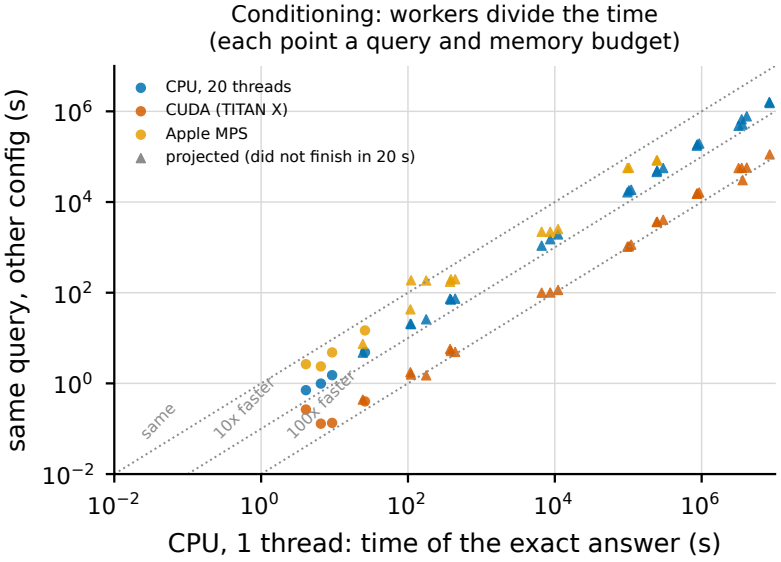


Figure 8: Each point is one conditioned query: its time on another configuration against its time on 1 CPU thread (triangles: projected from the measured rate). Workers divide the time almost uniformly, because instantiations are independent rows of one batched elimination.

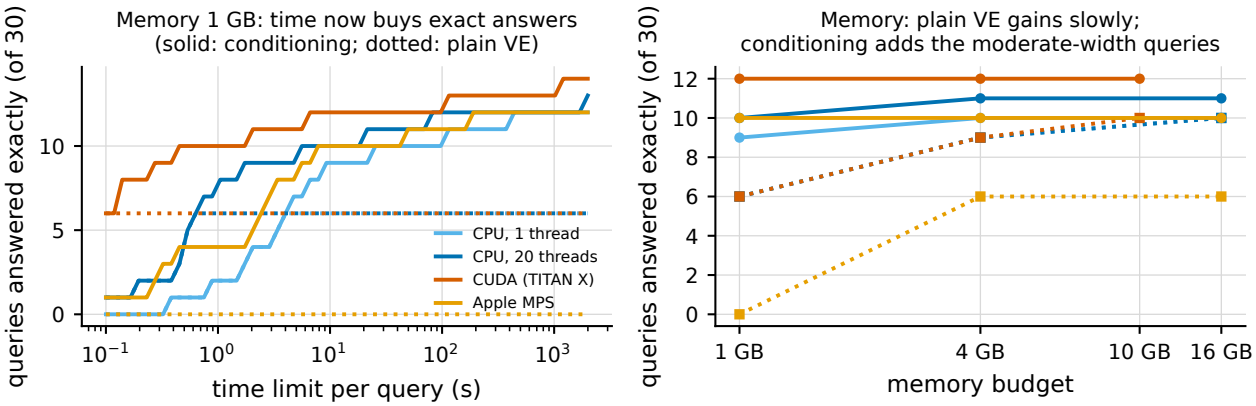


Figure 9: Left: at 1 GB, the number of exact answers grows with the time limit (solid), while plain VE (dotted) is flat; beyond 20 s the times are projected from the measured rate. Right: by memory budget, within 20 s.

Table 8: R4-1. Exact queries answered (of 10) on denser graphs (ER, 100 nodes, K = 3): plain VE vs conditioning within 20 s per query. Not finished: conditioning started (at most 10^6 instantiations) but stopped at 20 s, with the time it would need projected from its measured rate; over cap: more than 10^6 instantiations, not attempted

config	memory	avg degree	plain VE	conditioning <= 20 s	median s (answered)	cutset |C| (median / max)	not finished: n, projected s (median / max)	over cap
CPU, 1 thread	1 GB	3	6/10	8/10	2.54	0 / 4	2: 66.1 / 108	0
CPU, 1 thread	1 GB	4.5	0/10	1/10	9.27	13 / 18	3: 1.1e+05 / 3e+05	6
CPU, 1 thread	1 GB	6	0/10	0/10	–	18 / 21	2: 4.69e+05 / 9.27e+05	8
CPU, 1 thread	4 GB	3	8/10	9/10	3.14	0 / 2	1: 110 / 110	0
CPU, 1 thread	4 GB	4.5	1/10	1/10	10.5	11 / 16	6: 1.72e+06 / 8.35e+06	3
CPU, 1 thread	4 GB	6	0/10	0/10	–	16 / 19	2: 4.4e+05 / 8.72e+05	8
CPU, 1 thread	16 GB	3	9/10	9/10	3.08	0 / 1	1: 177 / 177	0
CPU, 1 thread	16 GB	4.5	1/10	1/10	10.5	10 / 15	7: 3.51e+06 / 3.98e+07	2
CPU, 1 thread	16 GB	6	0/10	0/10	–	15 / 18	3: 8.58e+05 / 2.1e+07	7
CPU, 20 threads	1 GB	3	6/10	9/10	0.535	0 / 4	1: 20.7 / 20.7	0
CPU, 20 threads	1 GB	4.5	0/10	1/10	1.52	13 / 18	3: 1.86e+04 / 5.61e+04	6
CPU, 20 threads	1 GB	6	0/10	0/10	–	18 / 21	2: 9.87e+04 / 1.95e+05	8
CPU, 20 threads	4 GB	3	8/10	10/10	0.568	0 / 2	0	0
CPU, 20 threads	4 GB	4.5	1/10	1/10	1.51	11 / 16	6: 2.64e+05 / 1.6e+06	3
CPU, 20 threads	4 GB	6	0/10	0/10	–	16 / 19	2: 9.38e+04 / 1.86e+05	8
CPU, 20 threads	16 GB	3	9/10	10/10	0.571	0 / 1	0	0
CPU, 20 threads	16 GB	4.5	1/10	1/10	1.51	10 / 15	7: 5.1e+05 / 6.27e+06	2
CPU, 20 threads	16 GB	6	0/10	0/10	–	15 / 18	3: 1.75e+05 / 3.9e+06	7
CUDA (TITAN X)	1 GB	3	6/10	10/10	0.0588	0 / 4	0	0
CUDA (TITAN X)	1 GB	4.5	0/10	2/10	2.93	13 / 18	2: 2.61e+03 / 4.07e+03	6
CUDA (TITAN X)	1 GB	6	0/10	0/10	–	18 / 21	2: 8.19e+03 / 1.63e+04	8
CUDA (TITAN X)	4 GB	3	8/10	10/10	0.061	0 / 3	0	0
CUDA (TITAN X)	4 GB	4.5	1/10	2/10	2.64	12 / 17	4: 2.99e+04 / 5.74e+04	4
CUDA (TITAN X)	4 GB	6	0/10	0/10	–	17 / 20	2: 7.81e+03 / 1.55e+04	8
CUDA (TITAN X)	10 GB	3	9/10	10/10	0.0614	0 / 2	0	0
CUDA (TITAN X)	10 GB	4.5	1/10	2/10	2.52	11 / 16	5: 3.04e+04 / 1.12e+05	3
CUDA (TITAN X)	10 GB	6	0/10	0/10	–	16 / 19	2: 7.63e+03 / 1.52e+04	8
Apple MPS	1 GB	3	0/10	9/10	2.03	3.5 / 8	1: 43 / 43	0
Apple MPS	1 GB	4.5	0/10	1/10	4.82	16.5 / 21	1: 174 / 174	8
Apple MPS	1 GB	6	0/10	0/10	–	21 / 24	1: 2.53e+03 / 2.53e+03	9
Apple MPS	4 GB	3	6/10	9/10	1.27	1 / 6	1: 188 / 188	0
Apple MPS	4 GB	4.5	0/10	1/10	5.24	14 / 19	3: 5.69e+04 / 8.35e+04	6
Apple MPS	4 GB	6	0/10	0/10	–	19 / 22	1: 2.21e+03 / 2.21e+03	9
Apple MPS	16 GB	3	6/10	9/10	1.26	1 / 6	1: 186 / 186	0
Apple MPS	16 GB	4.5	0/10	1/10	5.9	14 / 19	3: 5.63e+04 / 8.26e+04	6
Apple MPS	16 GB	6	0/10	0/10	–	19 / 22	1: 2.21e+03 / 2.21e+03	9

Reading. (1) At 1 GB, 30 queries: plain VE answers 6 (MPS: 0). Conditioning answers 9 within 20 s on 1 CPU thread, 10 on 20 threads, 12 on CUDA and 10 on MPS. With 1 hour per query (projected), the total reaches 12–14. (2) More memory means a smaller cutset but does not add exact answers beyond the degree-3 graphs. (3) On MPS, conditioning also removes the 16-dimension crash and the stricter memory guard (the MPS allocator does not reuse freed blocks within a query, so the guard charges more): 0/10 → 9/10 at 1 GB. (4) Degree ≥ 4.5 is out of reach: this is the exponential cost of K&F Thm 9.11, not an implementation limit.

3.2 E2. Anytime certified sampling

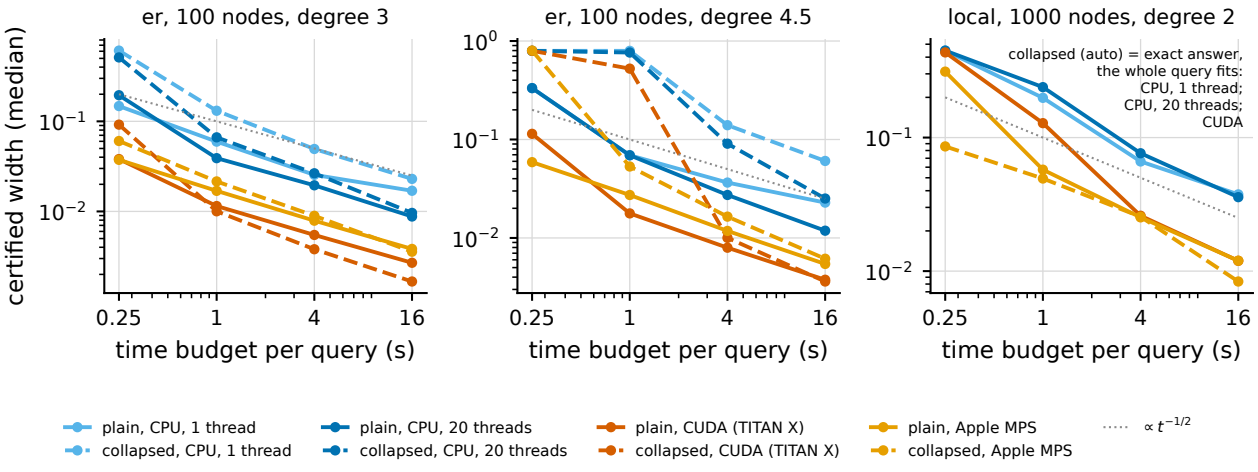


Figure 10: Median certified width (95%, all classes) by time budget, 10 queries per problem.

Table 9: R4-2. Certified sampling, anytime mode: median certified width (95%, all classes) by time budget; coverage of the exact answer where it is known

problem	config	mode	0.25 s	1 s	4 s	16 s	covers exact
er n=100 deg 3	CPU, 1 thread	plain	0.148	0.0596	0.0255	0.017	32/32
er n=100 deg 3	CPU, 1 thread	collapsed	0.61	0.131	0.0494	0.023	32/32
er n=100 deg 3	CPU, 20 threads	plain	0.195	0.039	0.0195	0.00878	32/32
er n=100 deg 3	CPU, 20 threads	collapsed	0.512	0.0665	0.0265	0.00963	32/32
er n=100 deg 3	CUDA (TITAN X)	plain	0.0381	0.0115	0.00548	0.00269	32/32
er n=100 deg 3	CUDA (TITAN X)	collapsed	0.092	0.01	0.0038	0.00167	32/32
er n=100 deg 3	Apple MPS	plain	0.0374	0.0169	0.0079	0.00383	24/24
er n=100 deg 3	Apple MPS	collapsed	0.0605	0.0215	0.00894	0.00357	24/24
er n=100 deg 4.5	CPU, 1 thread	plain	0.333	0.0691	0.0365	0.0229	4/4
er n=100 deg 4.5	CPU, 1 thread	collapsed	0.795	0.795	0.14	0.0605	4/4
er n=100 deg 4.5	CPU, 20 threads	plain	0.333	0.0691	0.0273	0.0118	4/4
er n=100 deg 4.5	CPU, 20 threads	collapsed	0.795	0.764	0.0907	0.0251	4/4
er n=100 deg 4.5	CUDA (TITAN X)	plain	0.114	0.0178	0.00796	0.00376	4/4
er n=100 deg 4.5	CUDA (TITAN X)	collapsed	0.796	0.524	0.01	0.00361	4/4
er n=100 deg 4.5	Apple MPS	plain	0.0589	0.0273	0.0118	0.00545	–
er n=100 deg 4.5	Apple MPS	collapsed	0.799	0.0532	0.0165	0.00615	–
local n=1000 deg 2	CPU, 1 thread	plain	0.448	0.198	0.0662	0.0375	40/40
local n=1000 deg 2	CPU, 1 thread	collapsed	2e-06	2e-06	2e-06	2e-06	40/40
local n=1000 deg 2	CPU, 20 threads	plain	0.448	0.239	0.0763	0.0358	40/40
local n=1000 deg 2	CPU, 20 threads	collapsed	2e-06	2e-06	2e-06	2e-06	40/40
local n=1000 deg 2	CUDA (TITAN X)	plain	0.434	0.128	0.026	0.0119	40/40
local n=1000 deg 2	CUDA (TITAN X)	collapsed	2e-06	2e-06	2e-06	2e-06	40/40
local n=1000 deg 2	Apple MPS	plain	0.312	0.0575	0.0253	0.012	40/40
local n=1000 deg 2	Apple MPS	collapsed	0.0858	0.0494	0.0255	0.00836	40/40

Reading. (1) Plain anytime sampling: the width falls like $1/\sqrt{t}$ on every configuration (dotted guide), and the GPU reaches a given width 4–16× sooner than CPU. Coverage was 584/584 runs where the exact answer is known (plain and collapsed). (2) Collapsed sampling: on the sparse 1000-node graph, `"auto"` finds that the whole query fits and returns the exact answer on CPU and CUDA (on MPS the stricter memory model leaves a small cutset: still the narrowest interval at 16 s). On the dense ER graphs it pays only on the GPU and at larger budgets (CUDA, degree 3, 16 s: 0.0017 vs 0.0027). On CPU it is 1–2.6× wider than plain sampling. The first version of the `"auto"` rule sized the cutset for one particle and never reached a checkpoint on dense graphs (width 1); the rule now sizes it for a batch of particles (commit `870d2ab`); the table shows the rerun. Recommendation: keep `collapse=None` as the default, and use `"auto"` on accelerators or when the query is close to fitting.

3.3 E3. Bounded conditioning

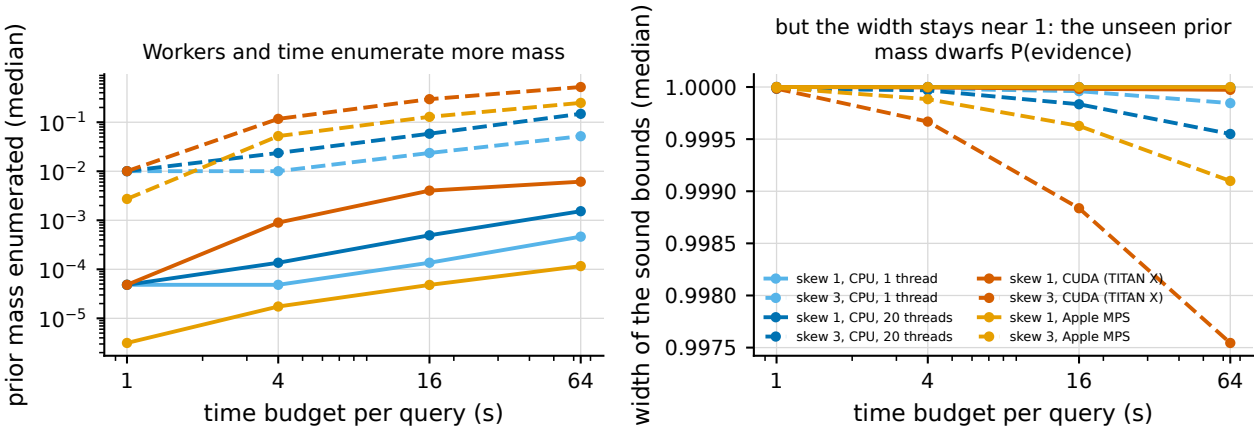


Figure 11: Bounded conditioning (degree 4.5, 1 GB, 5 queries; skew 3 multiplies the CPT logits by 3).

Table 10: R4-3. Bounded conditioning (sound bounds, no probability): median width by time budget; skew = factor on the CPT logits

config	skew	1 s	4 s	16 s	64 s	exact at largest	median prior mass covered
CPU, 1 thread	1	1	1	1	1	1/5	0.000464
CPU, 1 thread	3	1	1	1	1	1/5	0.0518
CPU, 20 threads	1	1	1	1	1	1/5	0.00153
CPU, 20 threads	3	1	1	1	1	1/5	0.148
CUDA (TITAN X)	1	1	1	1	1	1/5	0.00613
CUDA (TITAN X)	3	1	1	0.999	0.998	1/5	0.523
Apple MPS	1	1	1	1	1	1/5	0.000116
Apple MPS	3	1	1	1	0.999	1/5	0.248

Reading. Workers and time enumerate more prior mass: up to 52% on CUDA with skewed CPTs in 64 s. The bounds are sound, but their width $U/(A+U)$ stays near 1. The covered evidence mass $A = \sum_{c \in S} P(c, e)$ is at most $P(e)$, here about 10^{-2}–10^{-3} with 5 evidence variables, while the unseen mass is $U = 1 - \sum_{c \in S} P(c)$. So the width falls only when $U \lesssim P(e)$. This matches the condition in K&F §12.5: bounded conditioning works when a few instantiations carry most of the mass. One query in five (a small cutset) is exact.

4 Limits and next steps

- **The exponential wall stays.** Conditioning turns workers and time into exact answers only for moderate widths: the largest cutsets answered within 20 s have 5 variables ($3^5 = 243$ instantiations, about 6 s on CUDA or MPS); with 10 or more, projected times are 10^3 s and up, even on the GPU. Beyond that, the anytime certified interval is the right output: it uses every worker, and its guarantee is explicit.
- **Run queries in parallel.** LW gains only 1.26× from threads (round 3, per-node loop). A query-parallel executor (one query per worker) would use the cores fully for batches of queries. This is not implemented yet.
- **Tighter bounded conditioning** (ours, sketch, not implemented). For an unseen c, $P(c, e) \le W\, Q(c)$, where Q is the forward-sampling law with the evidence clamped, and $W = \prod_{e_i} \max_{pa} P(e_i \mid pa)$ (the LW weight bound, K&F Eq. 12.11). Replacing $U = 1 - \sum_S P(c)$ with $U_e = W(1 - \sum_S Q(c))$ is still sound with probability 1. It is tighter by a factor of about $W/P(e)$ in the required coverage, and enumerating in order of Q matches the bound. To be checked with `koller_claude` before implementing.
- **Portfolio fitting** (`fit` over table / MDN / flow candidates): the cost is the sum of the candidates. Successive halving (Hyperband, Li et al. 2018) and parallel candidates are the budget-aware versions; not implemented in this round.
- **Timing caveats.** Mac timings (round 3 and MPS) were taken on a shared machine with an idle gate, so they are conservative. Server timings are clean. Projected times assume a constant rate per instantiation.

References

- D. Koller, N. Friedman. *Probabilistic Graphical Models.* MIT Press, 2009 (§9.5 conditioning; §12.4 collapsed particles; §12.5 bounded conditioning).
- B. Bidyuk, R. Dechter. Cutset sampling for Bayesian networks. *JAIR* 28, 2007. https://arxiv.org/abs/1110.2740
- E. Horvitz, H. J. Suermondt, G. Cooper. Bounded conditioning: flexible inference for decisions under scarce resources. *UAI* 1989, pp. 182–193. https://arxiv.org/abs/1304.1512
- S. R. Howard, A. Ramdas, J. McAuliffe, J. Sekhon. Time-uniform, nonparametric, nonasymptotic confidence sequences. *Ann. Statist.* 49(2), 2021. https://arxiv.org/abs/1810.08240
- I. Waudby-Smith, A. Ramdas. Estimating means of bounded random variables by betting. *JRSS B* 86(1), 2024. https://arxiv.org/abs/2010.09686
- L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, A. Talwalkar. Hyperband. *JMLR* 18, 2018. https://arxiv.org/abs/1603.06560
- M. J. Kochenderfer et al. *Algorithms for Validation.* MIT Press, 2026 (Ex. 7.3, p. 196).

Reproduce

```
bash benchmarks/scaling/r3_rerun.sh && python benchmarks/scaling/r3_analyze.py
CONFIGS="cpu:1 cpu:20 cuda:4" bash benchmarks/scaling/r4.sh     # server
CONFIGS="mps:1" R=benchmarks/scaling/results/mac bash benchmarks/scaling/r4.sh
python benchmarks/scaling/r4_budget.py e2 --modes collapsed ...  # after 870d2ab
python benchmarks/scaling/r4_merge.py results server mac
python benchmarks/scaling/r4_analyze.py
cd benchmarks/scaling/report && tectonic -X compile report_r3.tex
```