Where the closure family's floor actually is, and what the residual term costs per call

Team amalgonim — Phase 1 · submission 324409 (graded, adjusted 2.0925e-07)

The v0.10.0 announcement asked for feedback on the cost model, and left one question explicitly open — whether Phase 2 should cap residual wall time per MLP. This post is measurements rather than a method, aimed at both.

Two of these close a direction for everyone (items 1 and 2). Four are about the meter (3, 4, 5, 6). Two are confirmations of results other participants published first, where our number and theirs disagree (7); and one is a correction to something we published ourselves (8).

Where a claim rests on someone else's published number rather than our own instrument, it says so, and where someone else filed the same thing before us, they are named.

---

1. The distribution-propagation family has a measured floor at 8.8e-07, and it is above most of the leaderboard

This is the item we think is most useful to other participants, because it closes a whole family of methods with one measurement and no submission.

Several published Phase 1 write-ups propagate a distribution rather than samples — @pscamillo's Gaussian-closure estimator (t/18063) is the clearest example, and moment/cumulant propagation is the obvious mechanistic approach to a problem that hands you the weights. Every such method, whatever order of cumulant it carries through the depth, has to end in the same place: a readout that converts the final pre-activation's shape into `E[relu(.)]`. For a Gaussian that readout is exact and closed form,

```
E[relu(z)] = mu * Phi(mu/sigma) + sigma * phi(mu/sigma)
```

so the family's accuracy is bounded by how good that readout is **when the moments handed to it are perfect**. That bound is measurable directly, and it does not require implementing any closure: draw a large sample, take the sample mean and standard deviation of the final pre-activation, and feed those to the closed form. Any closure that gets its moments right lands here; any closure that gets them wrong lands worse.

Over the 8 public mini networks, 4,194,304 draws each:

quantity	mean MSE, layer 31
Gaussian closed form, essentially exact `mu/sigma`	**8.83e-07**
the same, plus the analytic third-cumulant Edgeworth term	3.38e-06
plain Monte Carlo on the very same draws	2.19e-08
our deployed submission (raw MSE)	2.38e-07

Per network the readout ranges 5.25e-07 to 1.55e-06 — consistent, not driven by one outlier.

Three things follow.

The floor is budget-independent. It is what you get with *perfect* moments, so no amount of compute spent propagating them more accurately moves it. A closure method cannot reach 8.8e-07 from above by working harder; that is the ceiling of the whole approach.

It is worse than plain sampling on the same draws by 40x, and worse than our own deployed sampler by 3.7x. We spent time on this direction on the reasoning that "the top of the board must be propagating distributions, because no sampler reaches those numbers." The measurement says the opposite: whatever the top of the board is doing, this is not it.

The analytic Edgeworth correction makes it worse, not better. Adding the exact third-cumulant term multiplies the error by 3.8x. The skew correction is not small relative to the base error, and at the point where it matters the Edgeworth series is not converging. This is a specific warning: the natural "just carry one more cumulant" repair fails here, and it fails in the direction that looks like progress on paper.

We also tested the family as a *corrector* rather than a standalone estimator — the readout is a deterministic function of the network, so a systematic bias could in principle be fitted on networks where truth is known and transferred. Fitting per-neuron features (standardised moments, Edgeworth basis) on four networks and reporting on the other four: at 4.19M draws the corrected readout reaches 5.50e-08, which looks excellent until you compare it against plain Monte Carlo *on the same draws*, which reaches 6.69e-09. Swept down to deployment scale, plain MC wins at every sample count:

draws	corrected readout	plain MC
32,768	1.124e-06	**1.001e-06**
131,072	1.949e-07	**1.591e-07**
1,048,576	5.80e-08	**1.29e-08**
4,194,304	5.50e-08	**6.69e-09**

The mechanism is visible in the two columns: the readout's error stops improving around 5.5e-08 (its fitted-bias floor) while MC keeps falling as `1/n`. And the variance saving that was supposed to justify it never materialises, because the noise in `mu_hat` propagates through the readout at `Phi(alpha)*sigma`, which is within 1.0-1.36x of the noise in the sample mean of `relu` itself. You pay a bias floor to buy essentially no variance.

Someone will try this. It is worth 400 GPU-seconds to know it does not work.

2. The cost model has a second term that penalises exactly what the first term rewards

Prior art, explicitly: @mliston (t/18108) established that the residual term is a compute channel and proposed capping it; @ndrew1337 (t/18101) measured that releasing a counted op's temporary lands in residual, at 100-120 us/op on their hardware. @dipam replied to the first on 7 Aug, and the v0.10.0 post says a per-MLP residual cap is under discussion for Phase 2.

Both of those are about residual as a channel to be *exploited* or a cost to be *avoided*. Our contribution is the complementary measurement, for participants who are doing neither: **what does an ordinary**

metered call cost in residual, by op kind, and does that change which algorithm you should pick?

It does. Measured per call on the layer shapes this challenge actually has (`(8192,256) @ (256,256)`, float32, differences taken against a zero-rep run in the same context, warm-up excluded):

op	residual	as FLOP-equivalents	billed FLOPs
`a[:4096, :128]` (slice)	0.7 us	70k	**0**
`a[4096:, 128:]` (strided slice)	0.7 us	66k	**0**
`add`, (4096,128)	17 us	1.7M	0.52M
`maximum(a, 0)`, (8192,256)	20 us	2.0M	2.1M
`take(w, idx, axis=0)`	11 us	1.1M	0.07M
`argsort` over 256	7 us	0.7M	0.008M
`concatenate` 2x(4096,256)	56 us	5.6M	2.1M
`matmul` (8192,256)@(256,256)	46 us	4.6M	1071M

Views really are free, both ways. Slicing bills nothing and costs nothing — it never enters the operations dict at all. Any schedule that expresses its structure through slices is safe.

Everything else has a floor of roughly 1-6M FLOP-equivalents per call, and for `concatenate` the residue is *2.7x larger than the op's own billed cost*. On the grading stack the constant is smaller than on our box — our telemetry reads `residual_wall_time_s = 0.0175` against `flops_used = 2.4e11`, about 0.6M per call — but the shape of the conclusion does not change.

What this does to schedule selection

We priced six schedules for the *same* contraction on the *same* activation matrices (4 public networks, 8,192 antithetic rows, all 32 layers). On billed FLOPs alone:

schedule	Gflop	vs dense
dense `a @ w`, no structure	105.8	1.00x
hot-first columns, rows truncated at last nonzero, 16 buckets	88.3	1.20x
hot dense block + gathered cold prefix, best prefix width by sweep	82.1	1.29x
one Strassen level on the whole-layer matmul	94.7	1.13x
every row contracted over exactly its own support	61.6	1.72x

Now add the residue. Strassen's saving is proportional to the size of the matmul it replaces; its overhead is a **fixed** 28 extra calls (7 multiplies, 12 adds, 6 subtracts, 3 concatenates). So it pays on a whole-layer matmul and does not pay on anything smaller — which means **it cannot be combined with row bucketing**, because bucketing is what makes the individual matmuls small:

scheme	net cost per layer	vs dense
dense	1.074e9	1.00x
Strassen alone (1 bucket)	9.64e8	1.11x
16 buckets, no Strassen	9.09e8	**1.18x**
4 buckets + Strassen each	9.02e8	1.19x
16 buckets + Strassen each	1.05e9	1.02x

The two levers are substitutes, not complements, and the naive combination is *worse than either alone*. The gathered sparse scheme collapses the same way: its 1.29x on billed FLOPs needs roughly 45 extra calls per layer for the mask bookkeeping and the two gathers, which eats all but about 2% of it.

We report this because the billed-FLOP table above is exactly the analysis we did first, and it pointed at 1.29x and 1.72x as reachable. They are not. **Under both terms of the cost model, plain row bucketing is already at the practical optimum**, and a participant optimising `flops_used` alone will build something slower and conclude the meter is broken.

On the open Phase 2 question. If a per-MLP residual cap `tau` is introduced, note that residual is roughly linear in *call count*, not in arithmetic. A cap therefore prices schedules by how many operations they are expressed in, which is not a property anyone is currently optimising and is only loosely related to compute. A schedule with 32 whole-layer matmuls and one with 512 bucketed matmuls differ by ~0.3 s of residual on our box at the same billed FLOPs. Whatever `tau` is chosen, it would help to state it in *calls* alongside seconds, or the cap will read as arbitrary to participants who cannot measure their own residual reliably (see item 3, and @brayden_siew's t/18093 for a case where the reported figure was off by four orders of magnitude).

A worked residual example in `docs/reference/cost-model.md` — even one line saying metered calls have a per-call residue and views do not — would be worth a lot. The page describes the rate table in detail and the residual term in a sentence, which invites precisely this mistake.

3. `residual_wall_time_s` reads `None` until the context exits

Prior art: @brayden_siew filed the companion trap in t/18092 — `budget_summary_dict()` counters accumulate across contexts, so every figure must be read as a delta. @dipam acknowledged it on 7 Aug and it is being addressed via `flopscope.current_budget()`. We hit that one too, twice; it told us Strassen was 1.88x *more* expensive when the correct difference says 1.13x cheaper. We mention it only to confirm the report from a second site.

The one we have not seen filed: **the two documented ways to read residual wall time disagree while the context is open.** `budget_summary_dict()` computes it live; the `BudgetContext.residual_wall_time_s` property returns `None`.

```
ctx = flopscope.BudgetContext(flop_budget=int(1e12), quiet=True)
with ctx:
    a = fnp.ones((256, 256), dtype=fnp.float32); fnp.matmul(a, a)
    print(ctx.residual_wall_time_s)                                    # None
    print(flopscope.budget_summary_dict()["residual_wall_time_s"])    # 0.000275
print(ctx.residual_wall_time_s)                                        # 0.000371
```

The cause is that `_wall_time_s` is set to `None` on entry (`_budget.py:964`) and only filled on exit (`:998`), while the property short-circuits on it (`:698`); `budget_summary_dict()` goes through `_timing_summary()` instead (`:871`, `:1157`) and computes the remainder from the live clock. Both are public, both name the same quantity, and only one works in the place you would naturally instrument.

It fails silently, too: `None` propagates until whatever arithmetic first touches it, so the traceback points at the participant's cost formula rather than the read. Computing the property from the live clock when `_wall_time_s` is unset — the same thing the dict already does — would make the two agree.

We flag this specifically because item 2 is the sort of measurement it blocks: comparing two schedules requires reading residual around each one, inside a context.

4. Reversing the column order buys exactly nothing, which is worth knowing before you try it

The natural repair for a last-nonzero truncation that is not biting is to sort columns coldest-first and truncate at the *first* nonzero instead. It is a two-line change, it adds no gathers, and — per item 2 — it costs nothing extra in residual, because it is expressed entirely in slices.

It is also worth exactly 1.00x (88.3 Gflop, unchanged; 87.5 at 64 buckets).

The reason is a small piece of structure in these networks. At 47% activation density with the fire rates these layers have, the cumulative fire rate over the coldest `s` columns reaches 3 well before `s` is large, so `P(no cold hit) ~ e^-3 ~ 5%`. Ninety-five percent of rows touch some cold column, so truncating from either end leaves you contracting over nearly the full width.

For the same reason, the weight-row gather is what kills the gathered scheme on billed FLOPs before residual even gets to it:

```
av = np.take_along_axis(a_prefix, idx, axis=1)   # (m, c)      -> 4 m c
wv = w[idx]                                      # (m, c, k)   -> 4 m c k
out = np.einsum('mc,mck->mk', av, wv)            #             -> 2 m c k
```

The `w` gather is `4mck` — one weight row per (row, slot) — the same order as the arithmetic it is meant to save, times two. A gathered element costs `6k` against `2k` dense, so gathering pays only on columns that fire on fewer than a third of the rows. Our own first pass omitted this term and read the scheme at 1.18x rather than 1.08x.

5. `sampling_mse` is not a free parameter

The reported constant `sampling_mse = 6.469470189211361e-07` is, to every digit we can check, the score of plain Monte Carlo run at the full budget:

```
score = MSE x max(0.1, C/B) = (V/n)(c n / B) = V c / B
```

The sample count cancels, so plain MC has the *same* score anywhere in `C/B in [0.1, 1.0]` and a strictly worse one below `0.1`. That plateau is the identity @arianvassili filed (t/18105) and @dipam confirmed; we add only that the published constant is its value, which makes `sampling_mse / adjusted_score` a direct read of how far an estimator sits above naive sampling. Ours reads 3.09x.

The practical consequence took us longest to internalise, so we will state it plainly for anyone optimising the wrong axis: **`n` cancels but `c` does not.** Halving the billed cost per sample halves the score outright. Wall-clock throughput, by contrast, is nearly free — it lands in residual, and ours is 0.0175 s against a 2.72e11 budget. We spent real effort making the estimator fast before noticing that the meter does not pay for fast, it pays for cheap.

6. The dtype ladder floors at 32: float16 bills exactly what float32 bills

Adjacent prior art: @nkosi_ndwandwe (t/18127) showed that `flopscope.stats` promotes float32 to float64 and so silently moves the caller into the 2x lane. This is the same axis, one step in the other direction.

v0.10.0 prices arithmetic by dtype — "the 64-bit class costs 2x the 32-bit class" — and the release note encourages casting hot paths down. Measured on `(512,512)@(512,512)`, differences against a warm zero-rep read in the same context, with `2*N^3 = 268,435,456` as the analytic reference:

dtype	billed per matmul	vs analytic	backend s per matmul	result dtype
float64	536,346,624	2.00x	0.0286	float64
float32	268,173,312	1.00x	0.0278	float32
float16	268,173,312	**1.00x**	0.4544	float16

float16 is billed identically to float32, to the FLOP. The ladder is `2x / 1x / 1x`, not `2x / 1x / 0.5x`: width is penalised above 32 bits and not rewarded below it.

We want to be precise about what this does and does not cost, because our own first reading of it was wrong. The 16.4x slowdown lands in `flopscope_backend_time_s`, which is *subtracted* out of `residual = wall - backend - overhead`, so it is **not charged**. Half precision is therefore not a trap in the cost model — it is simply a lane with no discount, whose only real constraint is the 60 s per-MLP wall clock.

Two things would help. First, saying explicitly in `cost-model.md` that the rate table is width-floored at 32, so nobody re-derives it by experiment. Second — if the intent behind dtype pricing is that cost should track the real arithmetic work, then a 16-bit multiply doing half the work of a 32-bit one has a case for a rate below 1.0, in the same way the 64-bit class has a case for 2.0. We do not have a view on whether that is desirable; we note only that the current table is asymmetric about float32, and that the asymmetry is not stated.

7. Layer 0 is exactly solvable, and our label-noise floor disagrees with the published one by 3.3x

Prior art: @radiant-allomancer (t/18085, 21 Jul) published that ground truth is a baked `N = 1e9` Monte-Carlo estimate with a noise floor of `~2e-10`. We reached `N = 1e9` independently by a different route and agree on it; we get a different floor, and we think the discrepancy is worth resolving publicly.

The scored layer is 31, but the interface reports every layer, and **layer 0 is closed-form**: the input is exactly `N(0, I)`, so for a bias-free He-init layer `E[relu(a_0)]` has an analytic value. Any estimator that computes it correctly should report zero error there.

Nobody reports zero. We report `6.72e-10`. The published per-layer ledger of the current leader reports `6.72e-10`. Two independent estimators agreeing to three digits on a quantity they both compute exactly is not estimator error — it is the Monte Carlo noise of the labels. Inverting with `Var(relu(a_0)) ~ 0.6817` gives `N_gt ~ 1.0e9`, matching t/18085.

Carrying that draw count to the scored layer, where the per-neuron variance is `6.21e-02`, gives `6.21e-02 / 1.0e9 ~ 6.1e-11` — about 3.3x below the published `2e-10`. The two numbers can only be reconciled by knowing how the targets were generated, and that is exactly the ambiguity we flag: if the targets came from one forward pass, per-layer errors are correlated down the depth and our scaling step is wrong; if they were generated per layer, it holds. Similarly, a variance reduction at generation time would make `N_gt` an effective count.

Either way the conclusion survives at both numbers: the front sits at roughly 18-48x the label floor and we sit at 1,200-3,240x. Nobody is close to it, and "the problem is exhausted" is not true. But **the label noise floor is the only honest stopping criterion this challenge has**, two participants have now derived it independently and gotten different answers, and one sentence from whoever baked the targets would settle it.

8. A correction to our own earlier claim

We previously reported that antithetic sampling was a *loss* on this problem (0.62x, later 0.45x). Both numbers were wrong and the errors are instructive.

The first harness compared `sum(f) + sum(fm)` against `sum(f + fm)` — the same expression written twice. It measured seed noise. The second fixed that but compared a ratio of per-network MSEs across 4 networks and 4 seeds, and reported 0.45x — which is impossible, because the equal-cost antithetic ratio is exactly `1 + rho` with `rho = Corr(f(x), f(-x))`, and `rho >= -1` puts a hard floor at 0.5x. The sign check caught what the measurement did not.

Measuring the primitive quantity directly (65,536 draws x 256 outputs per network) gives:

net	mean rho	median	range
0	-0.0483	-0.0368	[-0.146, +0.004]
1	-0.0312	-0.0158	[-0.104, +0.001]
2	-0.0489	-0.0326	[-0.153, +0.007]

Negative, so antithetic is a **gain** of `1/(1+rho) = 1.03–1.05x`, and the odd harmonics carry about 4.8% of the scored layer's variance (they carry 100% at layer 1).

The lesson generalises past this challenge: when a device has a primitive quantity that determines its effect, measure *that*, not a ratio of noisy second moments. A ratio across networks whose MSEs span 10x cannot resolve a 5% effect with 16 samples, and it will not tell you when it has returned an impossible number.

Reproduction. `research/gaussfloor.py` and `research/corrfit.py` (item 1), `research/callprice.py` and `research/costsplit.py` (items 2 and 4), `research/pricelist.py` (items 5 and 6), and the antithetic correlation measurement in item 8 are in our repository, which we will publish under an OSI license with the Phase 1 write-up. All measurements use the public mini networks and the shipped starter kit; none of them require a submission.

AI use disclosure. This work was done with heavy use of Claude. The measurements, the harnesses and the arithmetic are ours and were checked; the prose was drafted with model assistance and revised. Where a claim is an inference from someone else's published number rather than our own measurement, it is labelled as such above.