PyLO

Towards Accessible Learned Optimizers in PyTorch

Paul Janson* · Benjamin Thérien* · Quentin Anthony · Xiaolong Huang · Abhinav Moudgil · Eugene Belilovsky

* Equal contribution

Mila · Concordia · Université de Montréal · EleutherAI

MLSys 2026 — May 2026

SECTION

Motivation & Barriers

Learned optimizers: motivation and systems gap

- Neural network training highly non-convex — Hand-designed optimizers (Adam, AdamW) . Not provably optimal for problems.
- Learned optimizers (LOs): Small MLP, Meta-trained to predict the parameter updates.
- VeLO (4000 TPU-months) — lower loss than tuned NAdamW, without HP tuning.

Why are LOs rarely deployed?

- Per-parameter MLP → memory-bandwidth bound.
- Un-fused implementation: $mn \times d_{\text{feat}}$ tensor materialized in HBM.
- JAX-only and tighly-coupled to meta-training pipelines.

Takeaway: Systems overheads limit accessiblity of learned optimizers.

Four barriers to learned-optimizer adoption

1. JAX-only ecosystem
Reference impl (`google/learned_optimization`) is JAX.
70% of the ML community uses PyTorch.

2. Coupled with meta-training
Repos focus on **training** new optimizers, not **applying** them.

3. No weight-sharing standard
No HuggingFace-like hub for optimizer weights.

4. Per-step overhead
LO step is a small MLP **per parameter** — naively 10–100× Adam.

Takeaway: We address these via PyTorch-native implementation without meta-training focus, and fused CUDA kernels.

SECTION

Library Design

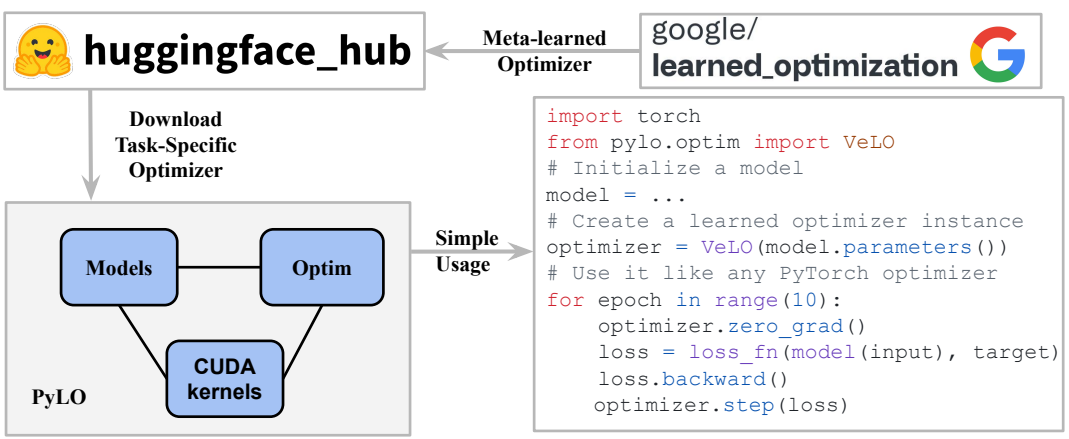
PyLO at a glance

Components

- Implements the `torch.optim.Optimizer` interface.
- HuggingFace Hub integration for meta-model weights (`from_pretrained`).
- Fused two-pass CUDA kernels: 86–88% step-time reduction vs un-fused PyTorch reference.

Composes with

- `torch.compile`, activation checkpointing
- LR schedules, decoupled weight decay
- DDP / FSDP, distributed optimizer step



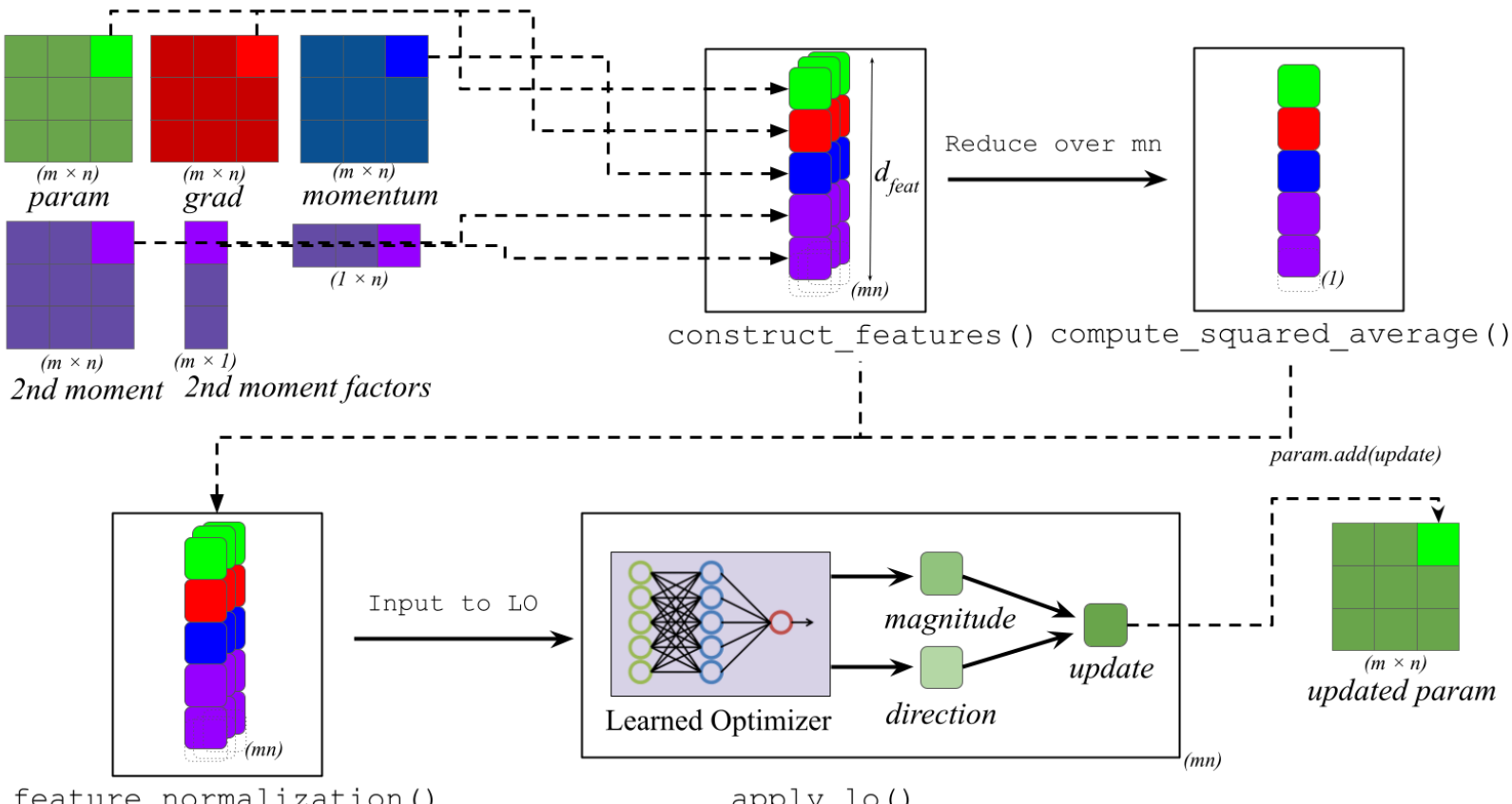
PyLO connects meta-learned optimizers with PyTorch training pipelines.

What it looks like to use PyLO

```
from pylo.optim import VeLO

optimizer  = VeLO(model.parameters(), hf_key="Belilovsky-Lab/velo")

# Compose freely with torch.optim utilities
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=N)

for x, y in loader:
    loss = model(x, y); loss.backward()
    optimizer.step(loss);
    scheduler.step();
    optimizer.zero_grad()
```

Takeaway: The learned optimizer conforms to the `torch.optim.Optimizer` interface and composes with standard PyTorch utilities.

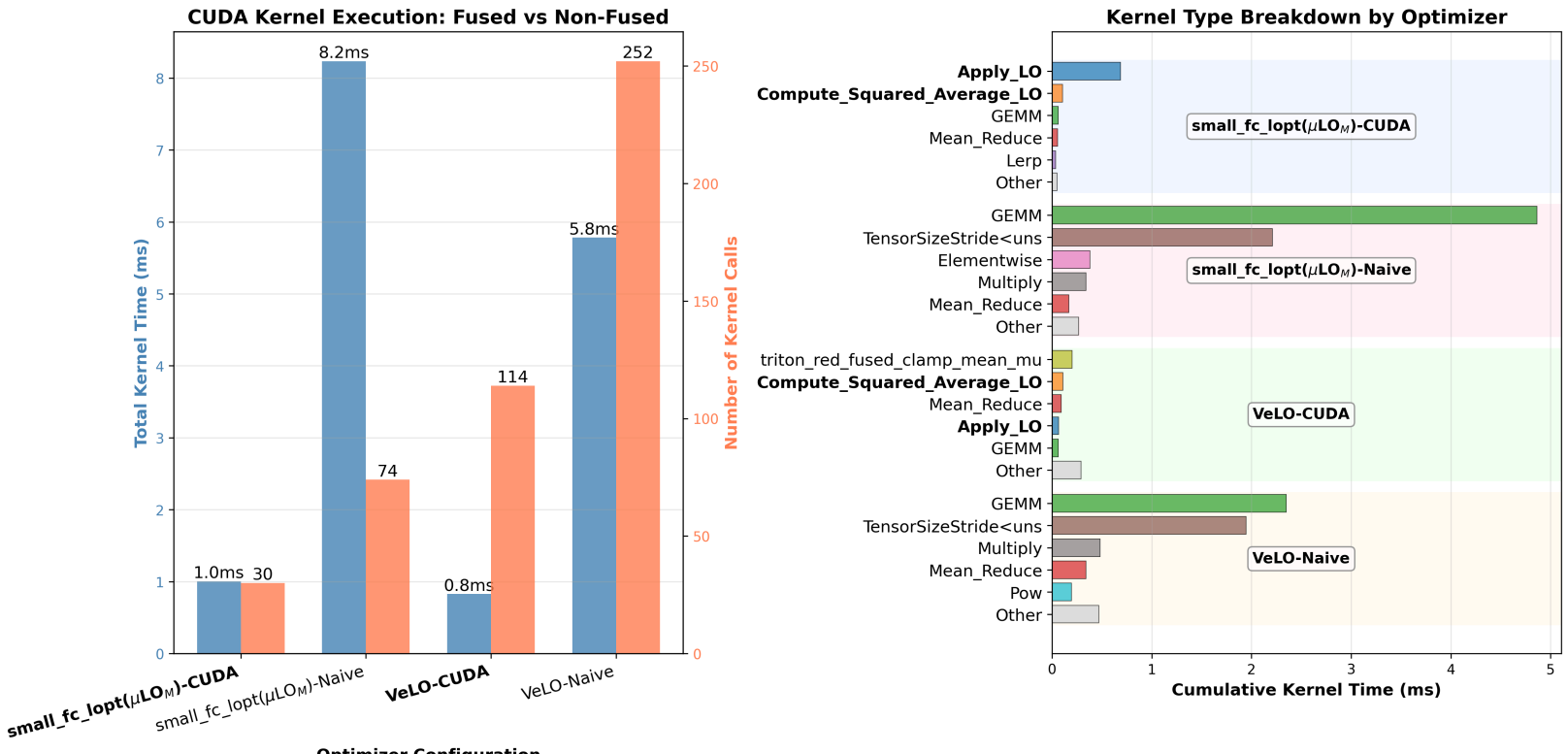
Anatomy of a learned-optimizer step



Why is the naive implementation so slow?

Per parameter tensor of size mn, the naive PyTorch path:

- Materializes a feature tensor of size $mn \times d_{\text{feat}}$ in **global memory**.
- Launches **74–252 kernels** — one per elementwise op, reduction, MLP layer.
- Memory-bandwidth bound.
- GPU compute idle.



Naive (top rows) — many small kernels; Fused (bottom) — few dense kernels.

Fused CUDA kernels: two passes, no intermediates

Kernel 1 — Statistics pass

- Compute features in registers — no materialization.
- Accumulate $\sum \text{feat}^2$ per thread.
- Warp-shuffle → block-reduce → atomic-add.
- Output: 30–39 scalars.

Kernel 2 — Apply pass

- Stats into SMEM; features in registers.
- Inline the MLP (weights via `__ldg`).
- Write parameter update only.

Memory hierarchy

Reg: feature vectors + MLP activations

SMEM: d_{feat} norm stats (broadcast)

HBM: only params, grads, accumulators

Wins

- 74–252 → **30–114** kernel launches
- Eliminates $mn \times d_{\text{feat}}$ feature tensor creation
- Eliminates multiple kernel launch overheads

SECTION

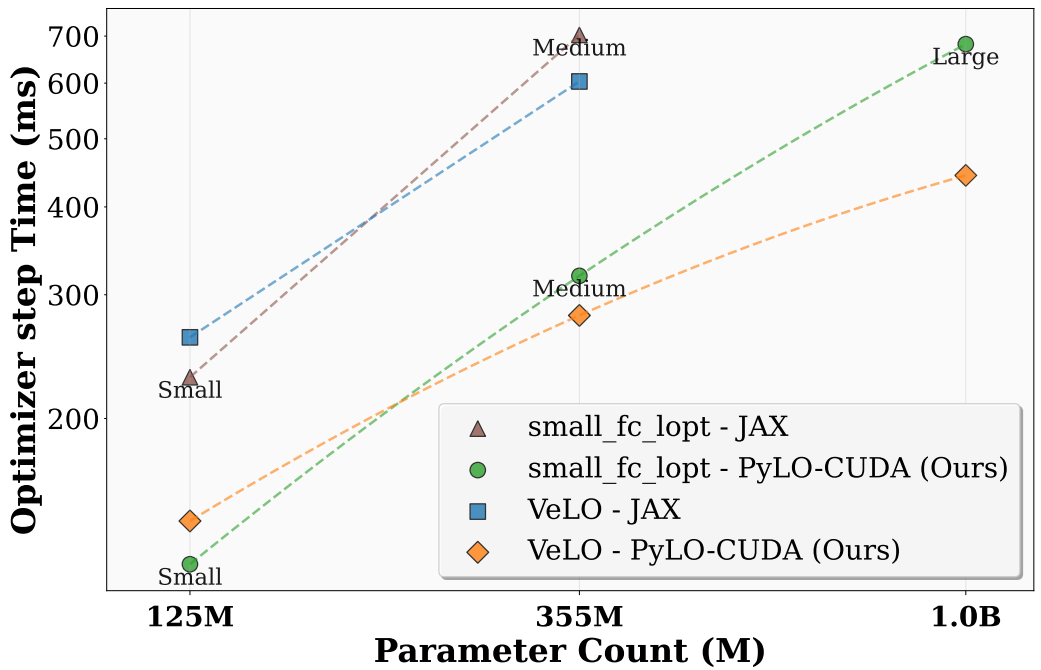
Evaluation

Step-time benchmarks: 86–88% reduction

Model	Optimizer	Opt step (ms)	Δ vs naive
ViT-B/16 (BS=32)	Adam	4.90	—
	Adafactor	18.99	—
	`small_fc_lopt` (naive)	756.80	—
	`small_fc_lopt` (CUDA)	**99.59**	−86%
	VeLO (naive)	585.11	—
	VeLO (CUDA)	**113.58**	−80%
GPT-2 355M (BS=4)	Adam	20.12	—
	Adafactor	35.11	—
	`small_fc_lopt` (naive)	2872.17	—
	`small_fc_lopt` (CUDA)	**319.14**	−88%
	VeLO (naive)	2378.93	—
	VeLO (CUDA)	**284.37**	−88%

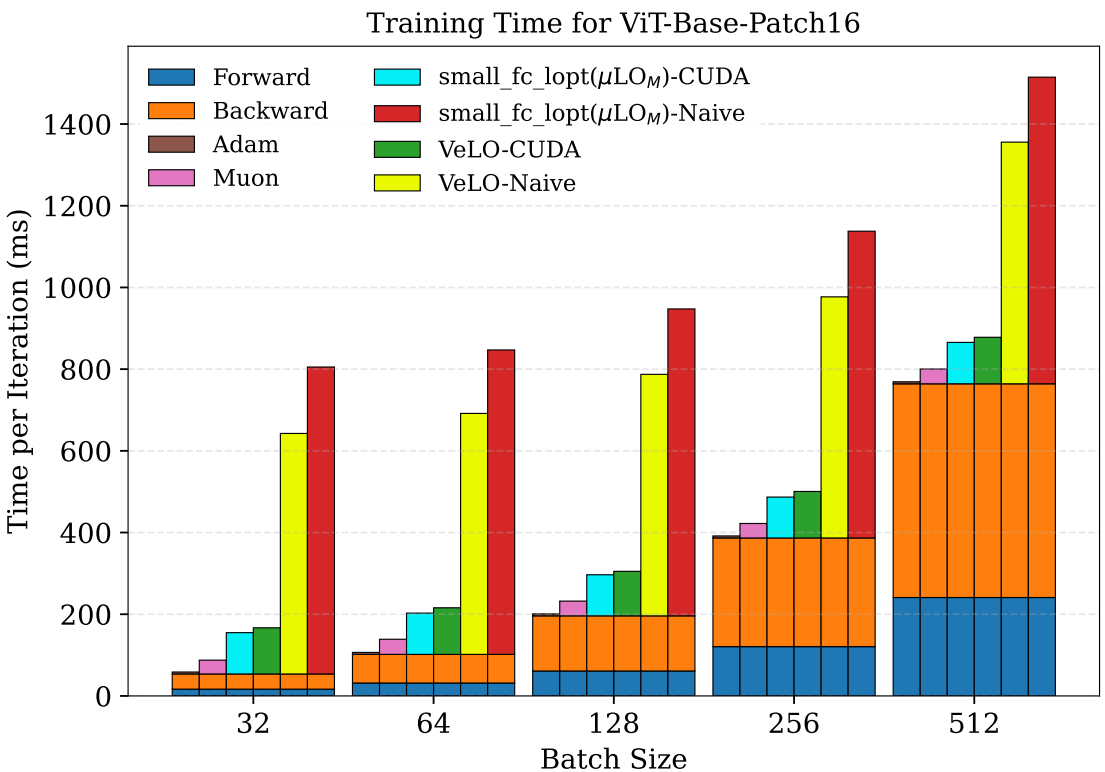
Takeaway: CUDA-fused step shrinks LO overhead by an order of magnitude.

Better scaling than the original JAX implementation



Takeaway: PyLO: 2× faster, no OOM at 1B params.

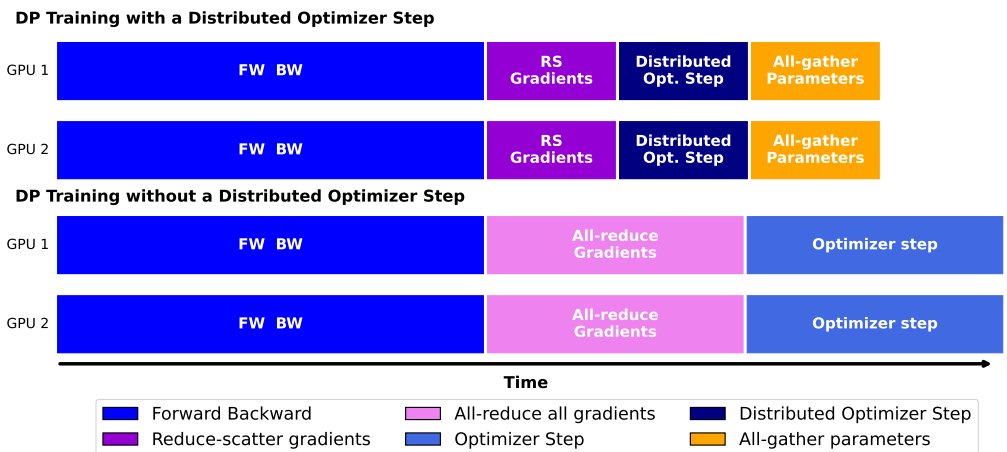
Overhead shrinks as batch size grows



ViT-B/16, single A100

- Optimizer step = fixed cost.
- LO overhead → negligible at large batch.
- PyLO makes LOs useful for large scale pretraining

Distributed optimizer step: more savings



All-reduce vs reduce-scatter optimizer step.

Training 125M GPT-2, with 4× H100

Optimizer	All-red	RS	Δ
`small_fc_lopt`	136.5	104.7	−31.9
VeLO	127.7	108.7	−19.0
Muon	106.9	98.1	−8.8
Adam	99.9	95.9	−4.0

Takeaway: Distributed overhead vs Adam: 9% / 13% for `small_fc_lopt` / VeLO.

Real-world results: ViT-B/16 + GPT-2 355M

ViT-B/16 on ImageNet-1K — 480 epochs

Optimizer	Top-1 acc
VeLO	**78.39%**
Adam + Cosine	77.22%
µLO$_m$	62.14%

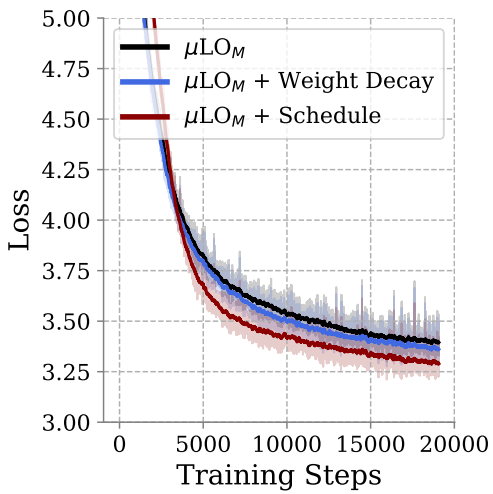
GPT-2 355M on FineWeb-EDU — 10B tokens

Optimizer	Loss
VeLO	**2.89**
Adam + Cosine	2.91
µLO$_m$	3.18

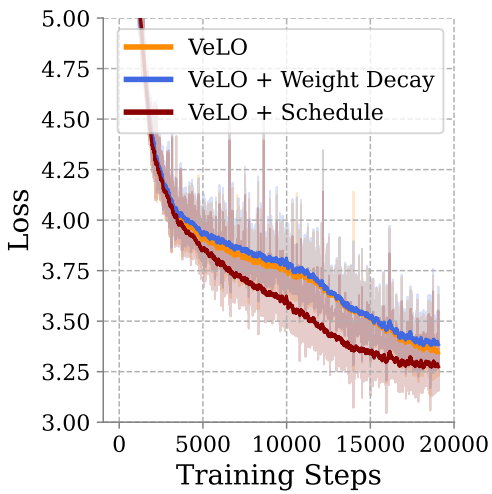
Findings:

- VeLO beats well tuned Adam with no LR schedule and tuning.
- µLO$_m$ shows good enough performance given short 1000 meta-train steps.
- µLO extends its performance with a schedule and WD.

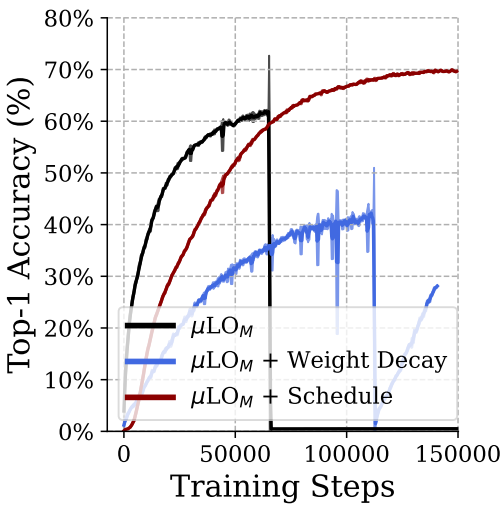
LR schedule & weight decay help learned optimizers



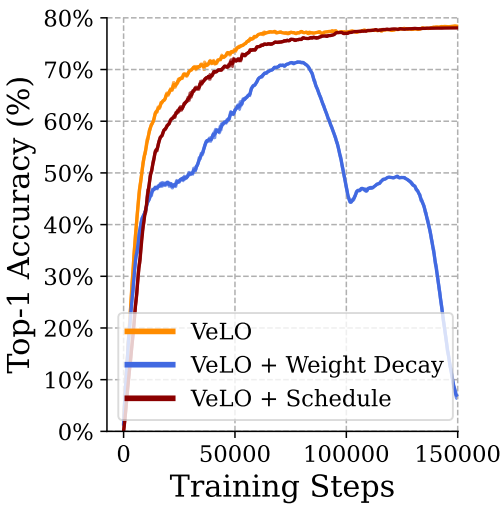
LM · μLO_m



LM · VeLO



ViT · μLO_m



ViT · VeLO

μLO_m gains a lot from schedule + WD; VeLO gains less — PyLO enables using both with a 5-line change.

Conclusion

We release PyLO a learned optimizer library in Pytorch

- `torch.optim.Optimizer` interface, HuggingFace Hub weights.
- 86–88% step-time reduction vs un-fused; 2× over JAX.
- Trains 1B GPT-2 on a single 80GB A100.
- Distributed step → 9–13% overhead vs Adam.
- Schedule + WD shows to improve learned optimizers and easy to use with PyLO.

Takeaway: Install now `pip install git+https://github.com/Belilovsky-Lab/pylo`

Mila

Thank you

Questions?

github.com/Belilovsky-Lab/pylo

SECTION

Appendix · Backup slides

Existing L2O libraries — feature comparison

Repo	Decoupled	PyTorch	HF Hub	CUDA	Paper
Open-L2O	✗	✓	✗	✗	Chen et al. ‘22
google/learned_optimization	✗	✗	✗	✗	Metz et al. ‘22
PyLO (ours)	✓	✓	✓	✓	*this work*

- **Decoupled** = optimizer code separated from meta-training code.
- Other repos focus on **training** LOs; PyLO focuses on **applying** them.

Learned-optimizer input features (per parameter)

Each parameter tensor has a feature vector of dimension $d_{\text{feat}} \approx 39$ (small_fc_lopt) or ≈ 30 (VeLO).

Raw accumulators

- parameter p
- gradient g
- momentum m (3 timescales)
- 2nd-moment v (3 timescales)
- row & column factors of v (factored Adafactor-style)

Derived features

- $g \times m$, $m \times \text{factors}$
- element-wise reciprocal-sqrt of normalizers
- power features (g^2, $|g|$, $\text{sign}(g)$)
- timescale-mixed ratios

Heavily ablated by Metz et al. 2022; diminishing returns past this set.

Kernel 1 — Fused statistics collection

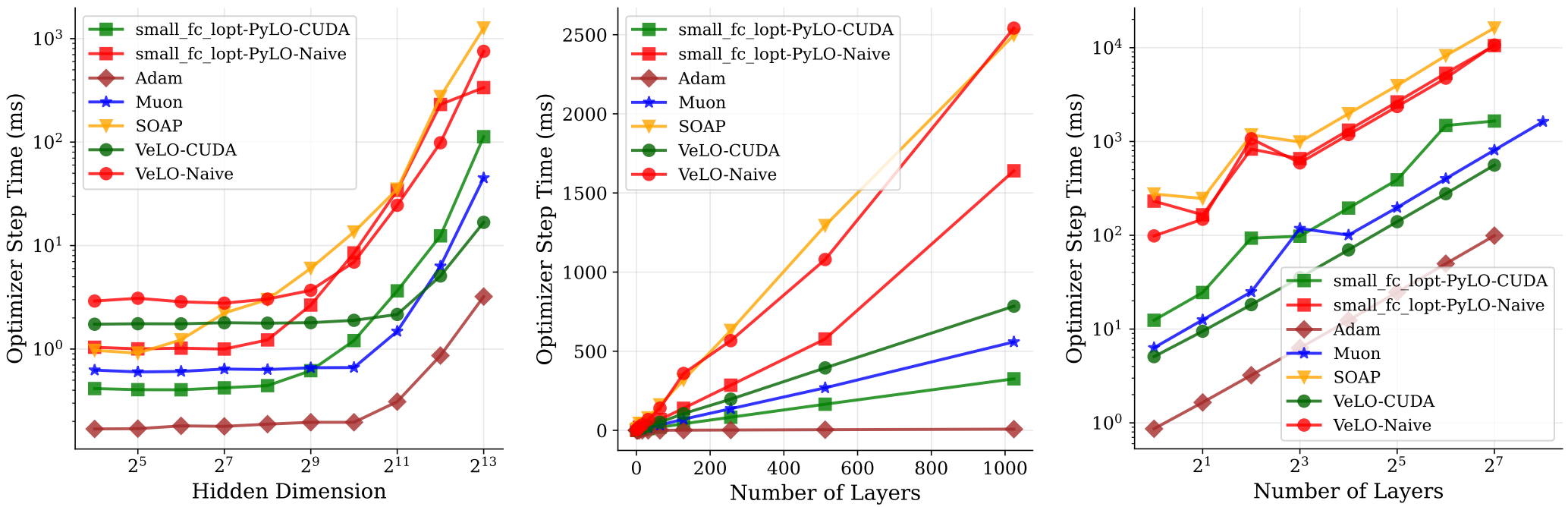
```
// Phase 1: compute features in registers, accumulate squared sum
Kernel_1(g, p, m, v, ...) {
    thread_accum[d_feat] = {0};            // register array
    for (i in grid_stride_loop) {
        features = compute_features(g[i], p[i], m[i], v[i], ...);
        thread_accum += features * features;
    }
    warp_reduce(thread_accum);             // __shfl_down_sync
    block_reduce(warp_results);            // shared memory
    atomic_add(global_stats, block_result);
}
```

- No feature tensor materialized — only d_{feat} scalars to global memory.
- Hierarchical reduction: register → warp → block → global.

Kernel 2 — Fused apply

```
// Phase 2: re-compute features, normalize, run MLP, apply update
Kernel_2(g, p, m, v, ..., global_stats) {
    __shared__ normalized_stats[d_feat];
    if (tid < d_feat)
        normalized_stats[tid] = rsqrt(global_stats[tid] / N);
    __syncthreads();
    for (i in grid_stride_loop) {
        features[d_feat]  = compute_features(g[i], p[i], m[i], v[i], ...);
        features         *= normalized_stats;            // broadcast SMEM
        hidden  = relu(Linear1(features));
        hidden  = relu(Linear2(hidden));
        dir, mag = Linear3(hidden);
        p[i]    -= dir * exp(mag * alpha) * beta;
    }
}
```

Scaling: synthetic MLP optimizees



(a) 1-layer MLP, varying width(b) width-256, varying depth(c) width-4096, varying depth

- CUDA implementation is **order-of-magnitude** faster than naive across all geometries.
- Approaches the envelope of traditional optimizers; missing points = OOM.

Sharded optimizer states (ZeRO-1 / FSDP A2A)

Optimizer	All-reduce	Reduce-scatter (Δ)	FSDP A2A (Δ)	Memory
`small_fc_lopt` CUDA	136.53	104.67 (−31.86)	110.53 (−26.00)	sharded
VeLO CUDA	127.67	108.71 (−18.96)	116.00 (−11.67)	sharded
Muon	106.92	98.10 (−8.82)	100.92 (−6.00)	sharded
Adam	99.93	95.93 (−4.00)	96.46 (−3.47)	sharded

- Reduce-scatter is **fastest**; FSDP A2A trades small overhead for **optimizer-state sharding**.
- Future: replace A2A with distributed normalization — only d_{feat} scalars exchanged.

Why memory bandwidth (and not FLOPs) is the bottleneck

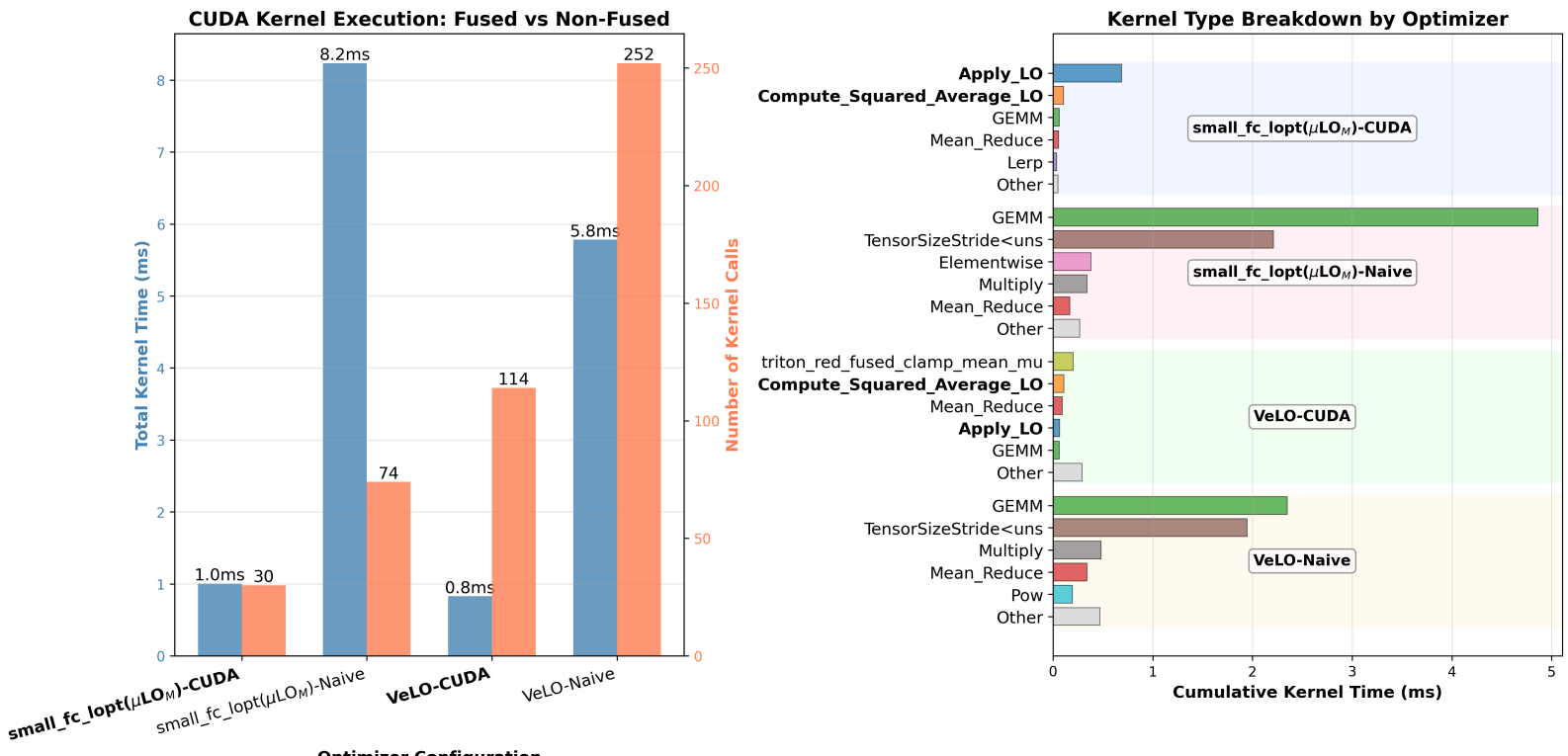
For a tensor of mn elements, the LO step reads/writes:

- **Inputs:** $p, g, m_1, m_2, m_3, v_1, v_2, v_3$, row/col factors → $\approx 10 \cdot mn$ words read.
- **Output:** updated p → mn words written.
- **MLP work:** a 4-layer MLP with $d_{\text{feat}} \approx 40, d_{\text{hidden}} \approx 32$ — very few FLOPs.

> Arithmetic intensity (FLOPs / byte) is **very low**.
> **Hide the MLP inside the streaming load** → memory-bandwidth-limited like Adam.

- Naive impl pays this bandwidth cost **many times** because of intermediates.
- Fused impl: each word of g, p, m, v is read **exactly twice** (Kernel 1 + Kernel 2).

Kernel-launch breakdown (full version)



Single MLP layer [1000×1000], no bias. Fusion applies only to the LO step — optimizee fwd/bwd unchanged.

Limitations and what's next

Current limitations

- LO step is still **slower** than Adam in absolute terms (5–10×).
- Kernels assume per-parameter feature dim ≲ SRAM/SM; very large d_{feat} needs new design.
- µLO$_{\text{m}}$ has limited training horizon — fixable by re-meta-training, not a library issue.

Future work enabled by PyLO

- **Hardware–optimizer co-design** — meta-train under a measured-cost objective.
- **Behaviour-cloned distillation** of expensive second-order optimizers (SOAP, Shampoo).
- **Sharded LO step** with d_{feat}-scalar all-to-all (vs full tensor).
- Community-shared LO weights on HuggingFace Hub.