A Artifact Appendix

A.1 Abstract

PyLO is a learned optimizer library that provides an accessible way to load and run pretrained learned optimizers in downstream tasks. It follows the standard `torch.optim.Optimizer` interface, enabling drop-in replacement of conventional optimizers such as Adam or AdamW with minimal changes to existing training code. PyLO also provides fast custom CUDA kernels that substantially reduce optimizer step time compared to naive PyTorch or JAX implementations. The core library is available at https://github.com/Belilovsky-Lab/pylo, and a companion repository of standard ML workflow examples is available at https://github.com/Belilovsky-Lab/pylo_examples. The latter provides reproducible training scripts for image classification (ViT-B/16 on ImageNet) and language model pre-training (GPT-2-style transformers), enabling direct comparison of learned optimizers against baselines such as AdamW with cosine annealing.

A.2 Artifact Checklist

- **Algorithm:** Learned optimizer inference (`small_fc_lopt`, VeLO, µLO) with CUDA-accelerated optimizer steps.
- **Program:** Python library (`pylo`) and example training scripts (`pylo_examples`).
- **Compilation:** CUDA toolkit required for custom kernel build; `CUDA_HOME` must be set. CUDA version of PyTorch must match `nvcc` version.
- **Binary:** No precompiled binaries are distributed; the package is built from source.
- **Data set:** ImageNet (for ViT experiments); standard language modeling datasets (for GPT experiments). Datasets are not bundled; users must supply them.
- **Run-time environment:** Linux, Python 3.11, PyTorch (CUDA-enabled), `huggingface_hub`. Optional: Weights & Biases for logging.
- **Hardware:** NVIDIA GPU (experiments in the paper use an A100 80 GB). CPU-only installation is possible but CUDA kernels will not be available.
- **Execution:** Single- or multi-GPU training via `torchrun`.
- **Metrics:** Optimizer step time (ms), training throughput (samples/sec), validation loss / accuracy.
- **Output:** Training logs (optionally synced to Weights & Biases), model checkpoints.
- **Experiments:** Step-time comparisons; image classification and language model pre-training comparisons.
- **How much disk space required (approximately)?** $\sim$5 GB (excluding datasets).
- **How much time is needed to prepare the workflow (approximately)?** 15–30 minutes (installation and environment setup).
- **How much time is needed to complete experiments (approximately)?** Step-time benchmarks: $<$1 hour. Full ViT or GPT training runs: several hours to days depending on hardware.
- **Publicly available?** Yes.
- **Code licenses:** BSD-3-Clause (`pylo`); Apache-2.0 (`pylo_examples`).

A.3 Description

A.3.1 How to Access

The PyLO library is publicly available at:

https://github.com/Belilovsky-Lab/pylo

The companion examples repository (training scripts for ViT and GPT workflows) is available at:

https://github.com/Belilovsky-Lab/pylo_examples

Documentation is hosted at https://belilovsky-lab.github.io/pylo. Pre-trained Learned optimizer weights are distributed via Hugging Face Hub and are downloaded automatically on first use (default `hf-keys` provided).

A.3.2 Hardware Dependencies

An NVIDIA GPU is required to build and use the custom CUDA kernels. The benchmarks reported in the paper were performed on a single NVIDIA A100 80 GB GPU. The library can be installed without CUDA kernels (CPU-only or standard PyTorch GPU), but optimizer step times will be significantly slower.

A.3.3 Software Dependencies

- Python 3.11
- PyTorch (CUDA-enabled build; CUDA version must match the installed `nvcc`)
- `huggingface_hub` (for automatic weight download)
- CUDA Toolkit (`CUDA_HOME` must be set for kernel compilation)
- *Optional:* Weights & Biases (`wandb`) for experiment logging
- *Optional:* Custom `mup` fork for µP / µLO support: https://github.com/bentherien/mup

A.3.4 Data Sets

- **ImageNet (ILSVRC-2012):** Required for ViT-B/16 image-classification experiments. Must be obtained independently and the path supplied via `--data-dir`.
- **Language modeling datasets:** Required for GPT pre-training experiments. Configuration is specified inside `./language_model_pretraining`.

A.4 Installation and Evaluation

A.4.1 Installation

Quick install (no CUDA kernels). This variant uses standard PyTorch operations and requires no compilation step:

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

Build from source with custom CUDA kernels (recommended). CUDA must be installed and `CUDA_HOME` must point to the CUDA installation. The CUDA version of PyTorch must match `nvcc`:

```
git clone https://github.com/Belilovsky-Lab/pylo
cd pylo
pip install .
python setup.py install --cuda
```

Alternatively:

```
pip install --no-build-isolation \
    --config-settings="--build-option=--cuda" .
```

Optional: MuP patch for µLO support. After installing the library, apply the MuP patch:

```
python -m pylo.util.patch_mup
```

Setting up the examples repository. A self-contained setup using Miniconda is provided:

```
install_dir=$PWD/pylo_install
mkdir $install_dir && cd $install_dir

wget https://repo.anaconda.com/miniconda/\
Miniconda3-py311_24.7.1-0-Linux-x86_64.sh
bash Miniconda3-py311_24.7.1-0-Linux-x86_64.sh \
    -b -p $PWD/miniconda3
source $PWD/miniconda3/bin/activate

pip install git+https://github.com/bentherien/mup.git

git clone https://github.com/Belilovsky-Lab/pylo.git
cd pylo && pip install .
python setup.py install --cuda
```

For Weights & Biases logging, set the following environment variables:

```
export WANDB_API_KEY=YOUR_KEY
export WANDB_PROJECT=pylo_examples
export WANDB_MODE=online
```

A.4.2 Basic Usage

PyLO optimizers conform to the `torch.optim.Optimizer` interface. The only difference from standard PyTorch optimizers is that for some optimizers the current loss value must be passed to `optimizer.step()`:

```
import torch
from pylo.optim import VeLO_CUDA

model = torch.nn.Linear(10, 2)
optimizer = VeLO_CUDA(model.parameters())

for epoch in range(num_epochs):
    optimizer.zero_grad()
    loss = loss_fn(model(inputs), targets)
    loss.backward()
    optimizer.step(loss)  # loss value required for VeLO
```

B Enhanced Performance on NVIDIA H100 GPUs

To further demonstrate the scalability of our approach on state-of-the-art hardware, we evaluate its performance on NVIDIA H100 GPUs, which offer advanced features. These accelerators are particularly beneficial for large matrix operations, such as those with dimensions 8192×8192—corresponding to the hidden size of a 70B-parameter transformer model.

In our enhanced implementation (denoted `small_fc_lopt-PyLO-CUDA++`), we introduce the following optimizations:

- Reduction in the number of global MLP reads to minimize HBM accesses;
- Asynchronous memory copies to overlap data movement with computation; and
- Full utilization of tensor cores and WGMMA instructions for high-throughput matrix multiplications on large tensors.

These modifications substantially reduce the optimizer step time, effectively closing the performance gap relative to state-of-the-art baselines for large-scale workloads. The results are summarized in Figure 1.

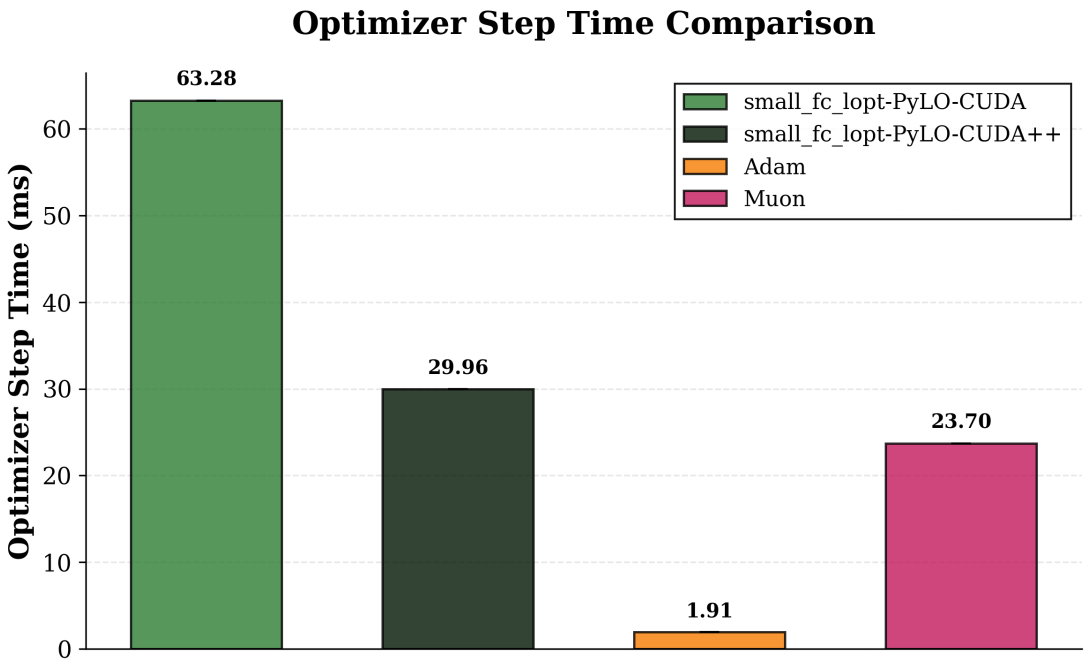


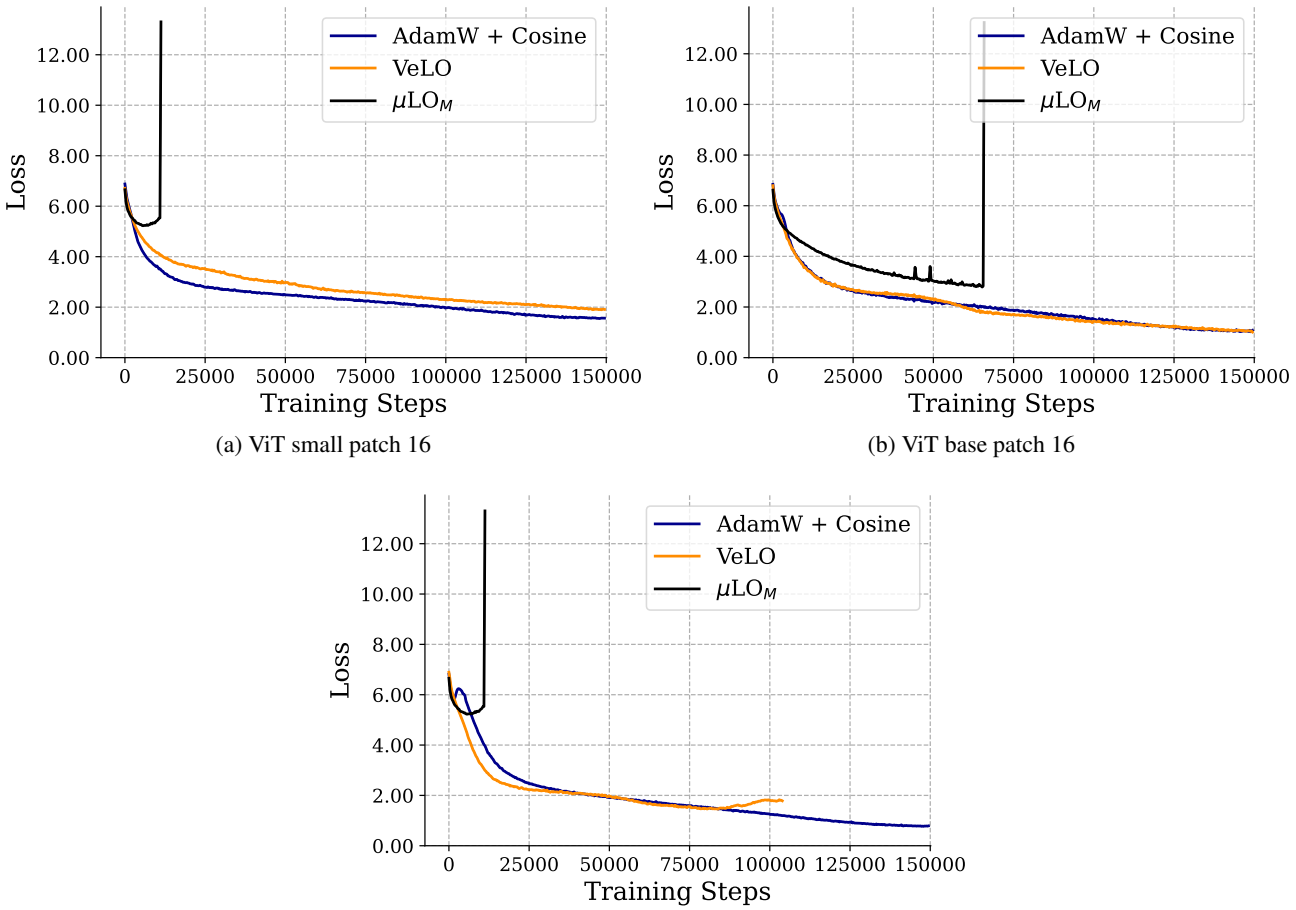
Figure 1. Optimizer step time comparison on NVIDIA H100 GPUs for 8192×8192 matrices. Our enhanced PyLO-CUDA++ implementation (dark green) achieves a $2.1\times$ speedup over the baseline PyLO-CUDA (light green). Muon (pink) and Adam (orange) given for reference

C Training Curves for the Experiments

C.1 Performance v.s. Steps

This section presents a comprehensive empirical evaluation comparing learned optimizers VeLO (**?**) and μLO (**?**) against the standard Adam optimizer (**??**) with cosine learning rate scheduling (**?**). Our evaluation spans Vision Transformer architectures of varying scales and GPT-2 models, extending beyond the main paper's validation accuracy results for ViT-Base/16 and final loss metrics for GPT-2 to provide detailed training dynamics across multiple configurations.

Vision Transformer Experiments: We evaluate three ViT configurations—Small, Base, and Large with patch size 16—on ImageNet-1k classification tasks. All configurations maintain consistent training protocols with 150,000 optimization steps and batch sizes of 4,096 across model variants, using hyperparameters detailed in Table 1. The extended evaluation



(a) ViT small patch 16

(b) ViT base patch 16

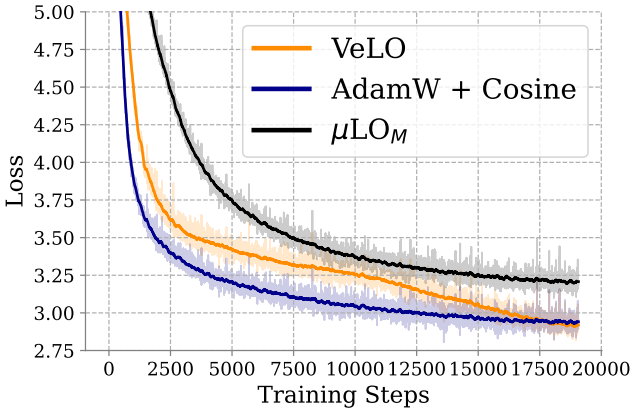
(c) ViT large patch 16

Figure 2. **Vision Transformer training dynamics.** Training loss curves for Vision Transformers trained on ImageNet-1k classification across three model configurations: (a) ViT-small with patch size 16, (b) ViT-base with patch size 16, and (c) ViT-large with patch size 16. We compare Adam with cosine learning rate scheduling (blue), VeLO learned optimizer (orange), and μLO learned optimizer (black) over 150,000 training steps with batch size 4,096. μLO exhibits early training instability across all scales, while VeLO demonstrates competitive or superior convergence properties, particularly evident in the base model configuration where it achieves lower final loss than the Adam baseline.

presented in Figure 4 reveals critical scale-dependent optimization characteristics. MuLO demonstrates pronounced early divergence patterns across both small and large model configurations, reinforcing our hypothesis that its constrained meta-training horizon limits generalization to extended optimization trajectories. In contrast, VeLO exhibits remarkable consistency in convergence properties across the full spectrum of model scales examined, maintaining stable training dynamics while achieving competitive performance metrics relative to the Adam baseline.

Language Modeling Experiments For GPT-2 pretraining on FineWeb-EDU data, we train models for 19,073 steps with batch size 512 and sequence length 1024. Complete hyperparameters are presented in Table 3, with full training curves shown in Figure 3. VeLO demonstrates competitive performance against Adam with cosine scheduling, while MuLO exhibits significantly degraded performance across the extended training horizon, consistent with the stability patterns observed in vision tasks.

C.2 Performance v.s. Wall-clock time



(a) 355M LM

Figure 3. **GPT-2 pretraining performance.** We pre-train decoder-only transformers of different sizes on a causal language modeling objective. We estimate gradients from 512 sequences of length 1024, resulting in a ∼0.5M token batch size per step. We train all models for 10B tokens of FineWeb-EDU data.

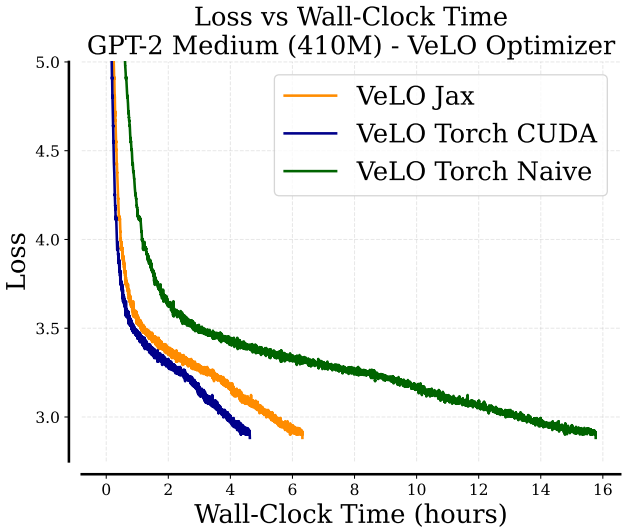


Figure 4. **GPT2 Medium wall-clock training time.**

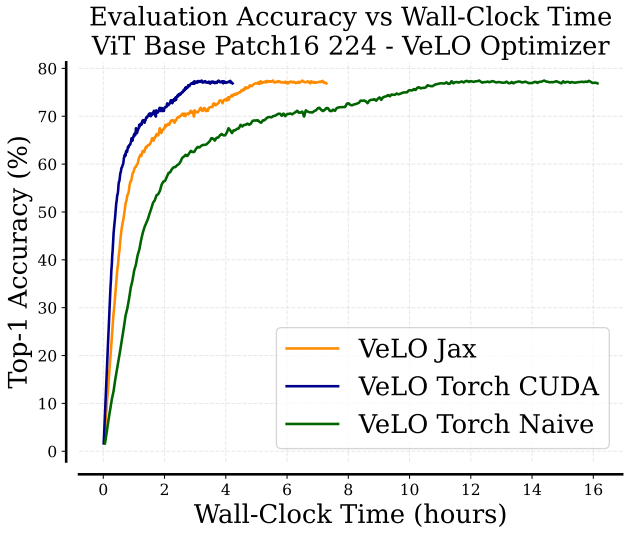


Figure 5. **ViT Base patch 16 wall-clock training time.**

D EXTENDED ABLATION STUDIES

This section presents comprehensive training curves across all hyperparameter configurations examined in our weight decay and cosine annealing ablations. Our analysis encompasses systematic sweeps of weight decay coefficients and maximum learning rates for cosine annealing schedules.

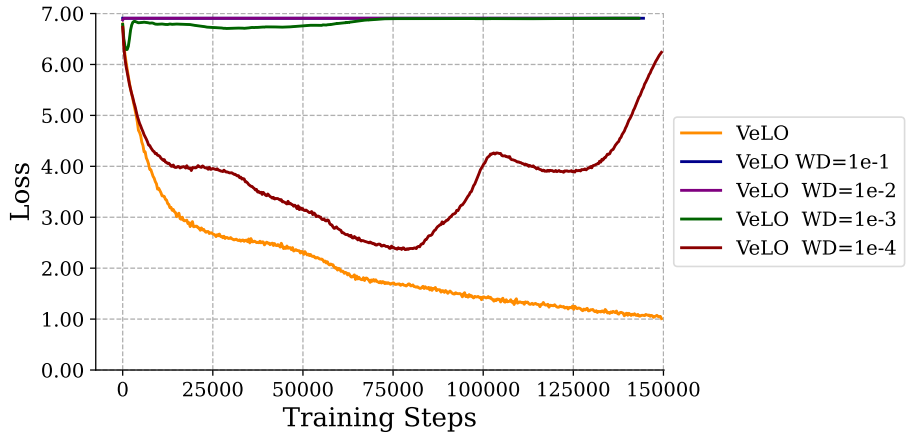
Vision Transformer Ablations Figures 6 and 7 demonstrate the variation of training dynamics with varying maximum learning rates and weight decay values for the ViT-B/16 model. The results reveal that VeLO exhibits minimal sensitivity to weight decay and scheduler modifications in this experimental setup, suggesting robust inherent regularization properties. Conversely, MuLO shows improved training horizon stability when augmented with cosine scheduling, enabling successful completion of the full 150,000 training steps. Weight decay modifications yield marginal improvements for MuLO across the evaluated parameter ranges.

Language Modeling Ablations

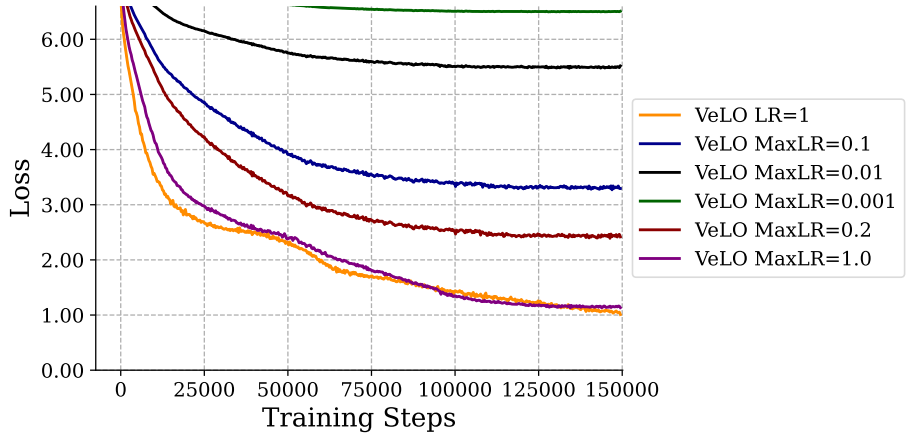
The language modeling experiments, presented in Figures 8 and 9, demonstrate more pronounced sensitivity to weight decay and scheduling compared to vision tasks. VeLO shows measurable performance improvements when augmented with weight decay and scheduling for specific parameter configurations. Similarly, MuLO benefits substantially from both weight decay regularization and cosine scheduling in the language modeling domain, suggesting that learned optimizers may benefit from scheduling and weight decay.

Architectural Configuration Effects

Figure 12 presents an additional ablation examining the impact of separate versus concatenated query-key-value (QKV) matrices on the optimizee's training loss. We observe that separating the QKV matrices in attention blocks improves performance across VeLO and μLO_M.

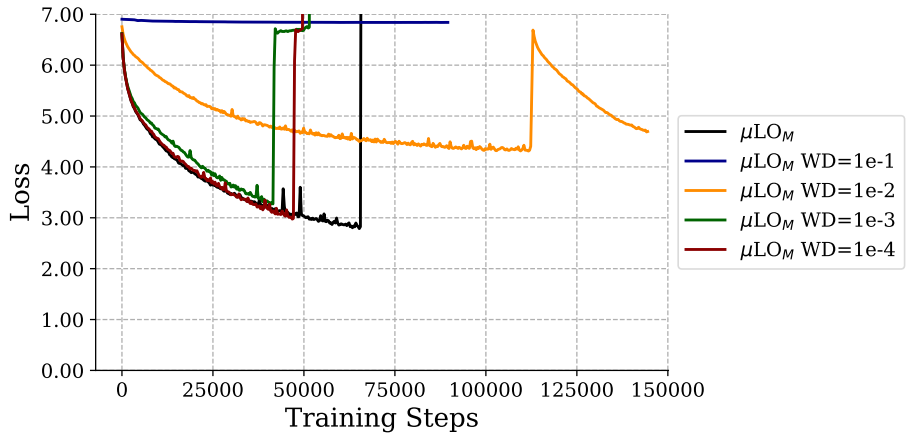


(a) VeLO Decoupled Weight decay ablation

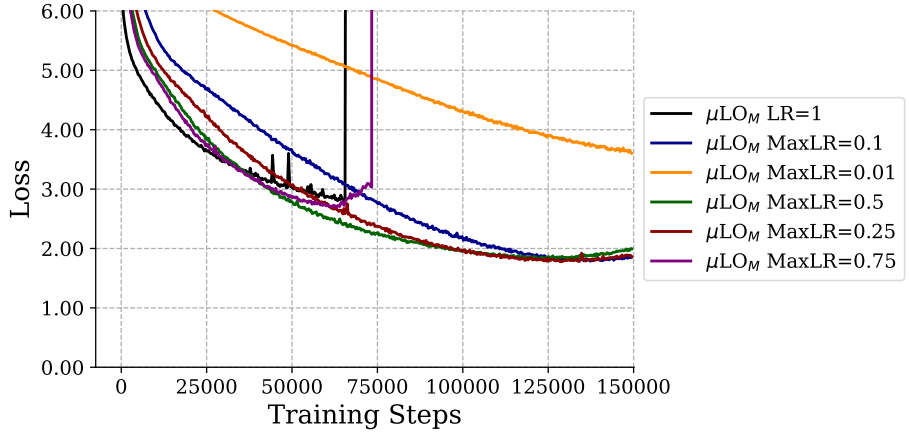


(b) VeLO Cosine annealing Ablation

Figure 6. **Training ViT-B-16 with VeLO on Imagenet 1k with decoupled weight decay and cosine annealing.** We see no improvement in using weight decay or Scheduler for VeLO in this setup

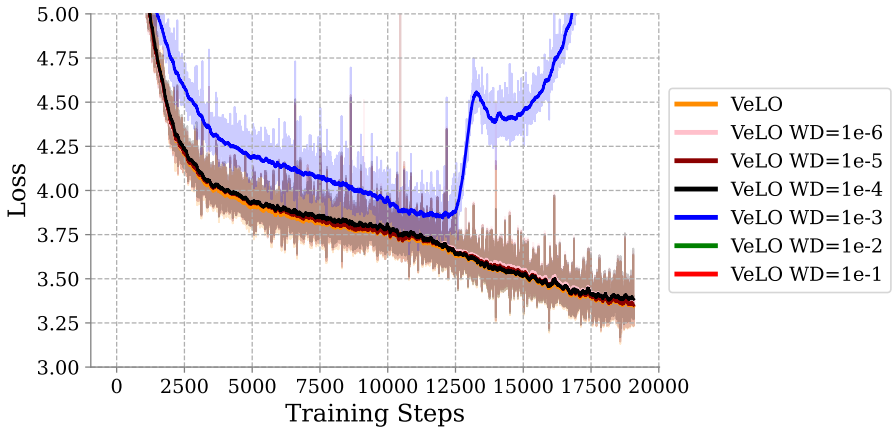


(a) μLO Decoupled Weight decay ablation

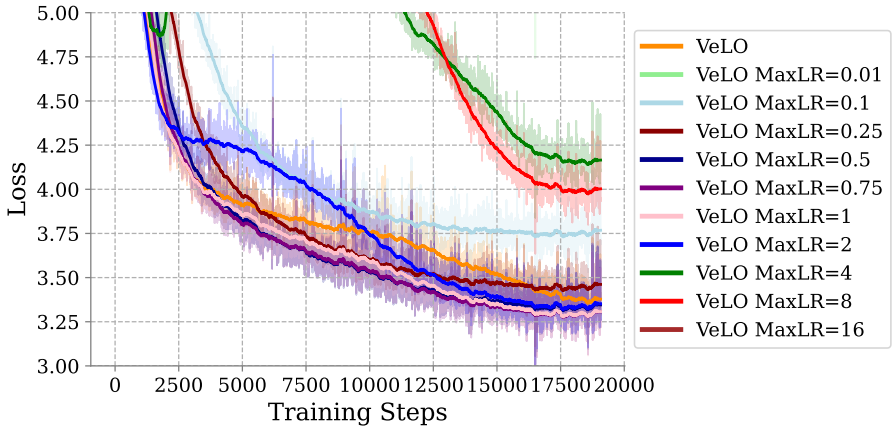


(b) μLO Cosine annealing Ablation

Figure 7. **Training ViT-B/16 with μLO on Imagenet1k with decoupled weight decay and cosine annealing.** We see that the training horizon of μLO is improved with using a scheduler that helps it to run up to 150000 training steps. We also see that weight decay did not have substantial improvement

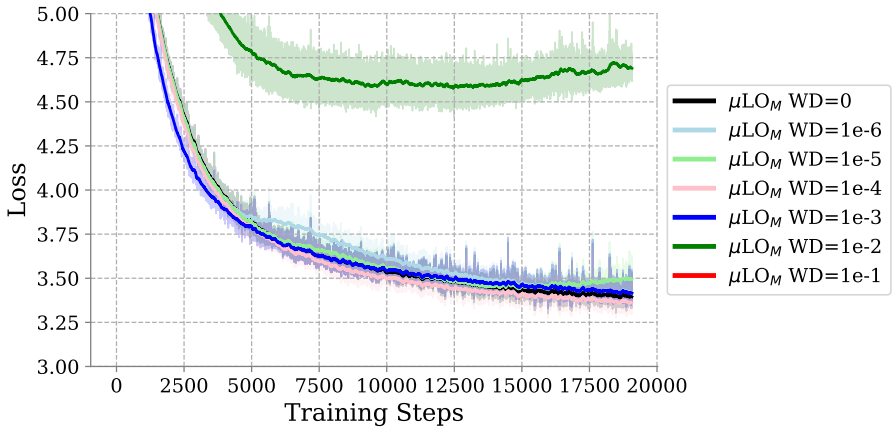


(a) VeLO Decoupled Weight decay ablation

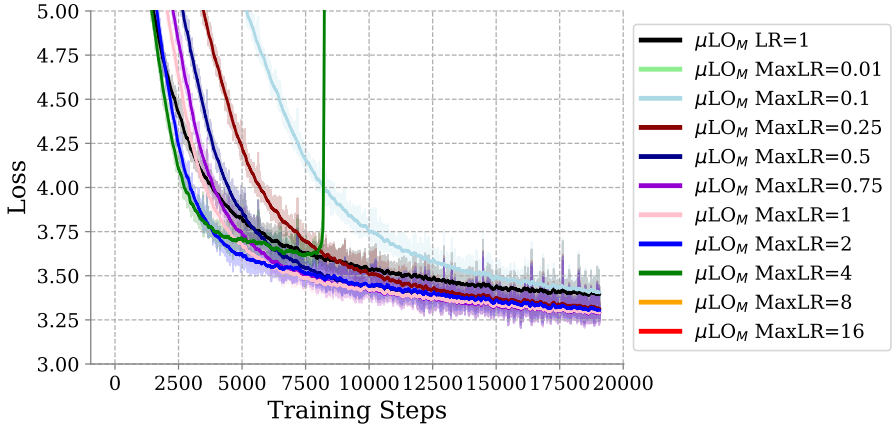


(b) VeLO Cosine annealing Ablation

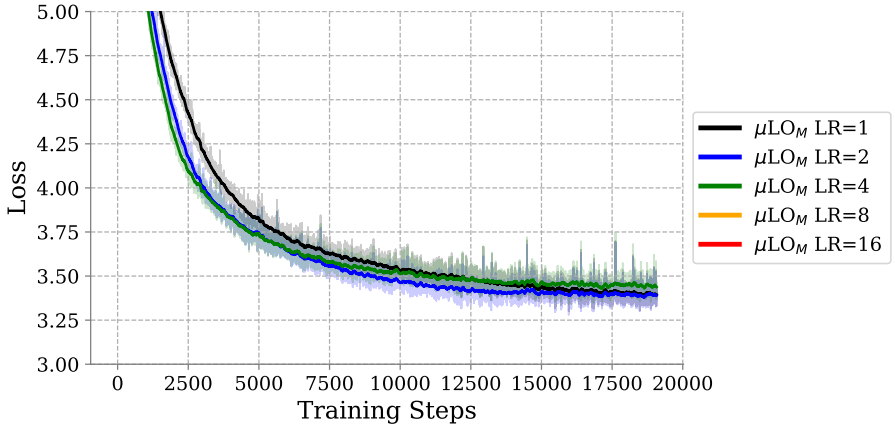
Figure 8. **GPT2 Pre-training VeLO ablations: decoupled weight decay and cosine annealing.** We pre-train decoder-only transformers of different sizes on a causal language modeling objective. We estimate gradients from 512 sequences of length 1024, resulting in a ~ 0.5 M token batch size per step. We train all models for $10B$ tokens of FineWeb-EDU data.



(a) μLO Decoupled Weight decay ablation

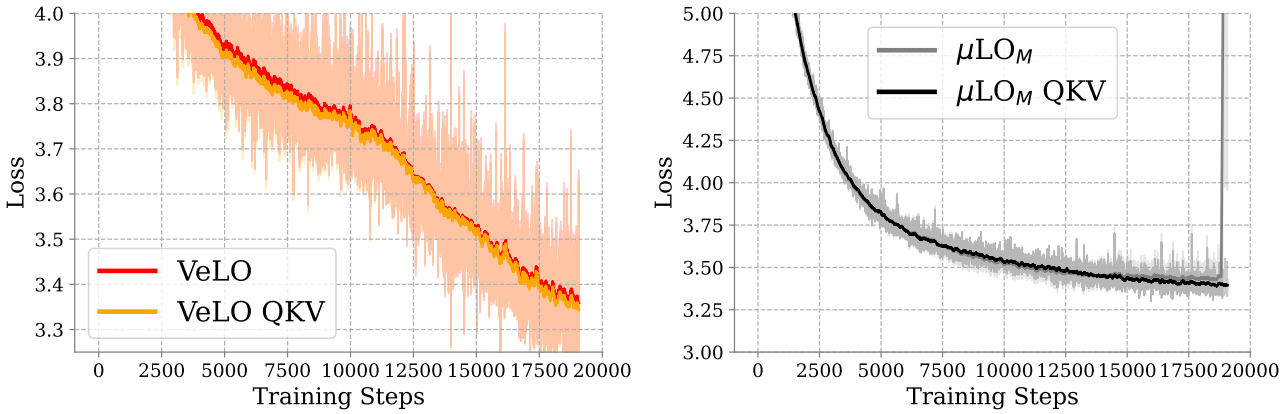


(b) μLO Cosine annealing Ablation



(c) μLO LR Tuning Ablation

Figure 9. **GPT2 Pre-training VeLO ablations: decoupled weight decay, LR-tuning, and cosine annealing.** We pre-train decoder-only transformers of different sizes on a causal language modeling objective. We estimate gradients from 512 sequences of length 1024, resulting in a ~ 0.5 M token batch size per step. We train all models for $10B$ tokens of FineWeb-EDU data.



(a) VeLO

(b) μLO

Figure 10. **Separate QKV weight ablation.** We ablate using concatenated or separate QKV matrices in attention layers. From the perspective of the learned optimizer, these two settings will be treated differently. We observe that performance improves for both μLO$_M$ and VeLO when QKV matrices are separated.

E HYPERPARAMETERS FOR EXPERIMENTS

Table 1. **Vision Transformer Training Hyperparameters.**

Description	Value
Model Architecture	
Model	ViT-Base/16
Image Size	224×224
Patch Size	16×16
Training Configuration	
Batch Size	4096
Training Epochs	480
Optimizer (for baseline)	Adam
Learning Rate (η)	4×10^{-3}
LR Scheduler	Cosine
Warmup Epochs	32
Weight Decay	0.03
Gradient Clipping	1.0
Data Augmentation	
Crop Percentage	0.95
Random Horizontal Flip	0.5
Mixup	0.1
CutMix	1.0
AutoAugment	rand-m7-mstd0.5
Regularization	
Dropout Rate	0.1
Stochastic Depth	0.1

Table 2. **Vision Transformer Model Variants Architecture.**

Description	Value
ViT-Small/16	
Parameters	22.05M
Hidden Dimension (d_{model})	384
MLP Hidden Dimension	1536
Number of Heads	6
Number of Layers	12
Patch Embedding Dim	384
ViT-Base/16	
Parameters	86.57M
Hidden Dimension (d_{model})	768
MLP Hidden Dimension	3072
Number of Heads	12
Number of Layers	12
Patch Embedding Dim	768
ViT-Large/16	
Parameters	307.40M
Hidden Dimension (d_{model})	1024
MLP Hidden Dimension	4096
Number of Heads	16
Number of Layers	24
Patch Embedding Dim	1024

Table 3. **GPT-2 Training Hyperparameters.**

Description	Value
Model Configuration	
Base Model	GPT-2
Architecture Type	Decoder-only Transformer
Attention Mechanism	separate_kqv
Sequence Length (T)	1024
Vocabulary Size	50257
Training Configuration	
Batch Size	512
Training Iterations	19073
Optimizer (for baseline)	Adam
Learning Rate Scheduler (for baseline)	Cosine
Warmup Iterations	381
Regularization	
Attention Dropout	0.1
Embedding Dropout	0.1
Residual Dropout	0.1
Initialization	
Muon Initialization	Enabled
Weight Initialization	Standard GPT-2

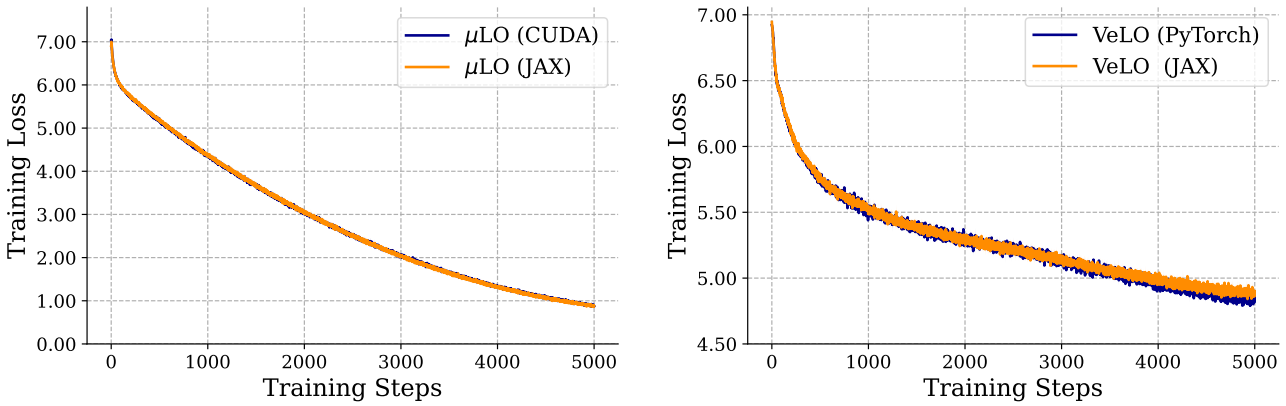
Table 4. **GPT-2 Model Variants Architecture.**

Description	Value
GPT-2 Small	
Parameters	$\sim$ 36M
Hidden Dimension (d_{model})	768
Number of Heads	12
Number of Layers	12
Head Dimension	64
FFN Hidden Dimension	3072
GPT-2 Medium	
Parameters	$\sim$ 345M
Hidden Dimension (d_{model})	1024
Number of Heads	16
Number of Layers	24
Head Dimension	64
FFN Hidden Dimension	4096
GPT-2 Large	
Parameters	$\sim$ 762M
Hidden Dimension (d_{model})	2048
Number of Heads	32
Number of Layers	16
Head Dimension	64
FFN Hidden Dimension	8192

F VALIDATION OF PYLO IMPLEMENTATION AGAINST ORIGINAL JAX CODEBASE

To validate the correctness of our implementation of the popular learned optimizers in PyLO, we conducted a direct comparison with the original JAX implementation from Google's learned optimization(**?**) codebase. We evaluated both μLO(**?**) and VeLO(**?**) algorithms on a simplified image classification benchmark using ImageNet resized to 64×64 pixels with a 3-layer MLP architecture (width 128). The comparison spans the first 5,000 training steps to assess implementation accuracy in a practical and economical setup.

The results demonstrate strong agreement between the two implementations across both optimizers. As shown in Figure 11, the training curves exhibit nearly identical convergence patterns across all 5 different runs. Minor variations arise from inherent differences in how PyTorch and JAX compute gradients. Accounting for this, we confirm that our PyTorch implementation faithfully reproduces the behavior of the original JAX code. We further plan to compare both implementations across a range of models and tasks.



(a) μLO implementation comparison

(b) VeLO implementation comparison

Figure 11. Training curve comparison between original JAX implementation and PyLO library for μLO and VeLO optimizers on ImageNet 64×64 classification task using a 3-layer MLP (width 128). Both implementations show nearly identical convergence behavior over 5,000 training steps, validating the correctness of our implementation.

G EXTENDED PER TIME STEP RESULTS

Model	Optimizer	Batch Size	Samples/sec	Step time (ms)	Fwd (ms)	Bwd (ms)	Opt step (ms)	Params (M)
ViT-B-16	AdamW	32	524.09	60.55	17.52	38.13	4.90	86.57
	Adafactor		425.51	74.69			18.99	
	small_fc_lopt (naive)		39.36	812.51			756.80	
	small_fc_lopt (CUDA)		205.59	155.16			99.59	
	VeLO (naive)		49.73	644.45			588.80	
	VeLO (CUDA)		191.18	168.61			112.97	
ViT-S-16	AdamW	32	1,116.12	28.17	7.65	18.27	2.25	22.05
	Adafactor		711.97	44.47			18.53	
	small_fc_lopt (naive)		109.03	293.01			266.90	
	small_fc_lopt (CUDA)		382.73	83.28			57.45	
	VeLO (naive)		94.84	337.68			311.77	
	VeLO (CUDA)		242.96	132.49			106.57	
ViT-L-16	AdamW	32	175.95	181.36	52.18	113.89	15.28	304.33
	Adafactor		156.64	203.34			37.23	
	small_fc_lopt (naive)		11.64	2,748.52			2,582.30	
	small_fc_lopt (CUDA)		70.69	451.69			285.80	
	VeLO (naive)		15.41	2,077.28			1,911.20	
	VeLO (CUDA)		76.74	418.29			252.21	
GPT2-125 M	AdamW	4	17.80	224.72	73.20	144.24	7.29	125.26
	Adafactor		17.02	235.07			17.76	
	small_fc_lopt (naive)		3.38	1,182.33			964.89	
	small_fc_lopt (CUDA)		11.71	341.46			124.09	
	VeLO (naive)		3.48	1,148.81			931.37	
	VeLO (CUDA)		11.25	355.49			138.05	
GPT2-410 M	AdamW	4	6.56	609.83	197.41	392.29	20.12	355.92
	Adafactor		6.40	624.64			35.11	
	small_fc_lopt (naive)		1.16	3,461.96			2,872.17	
	small_fc_lopt (CUDA)		4.40	908.69			319.14	
	VeLO (naive)		1.34	2,976.82			2,387.12	
	VeLO (CUDA)		4.58	873.41			283.71	
GPT2-1B	AdamW	4	2.88	1,387.51	442.49	896.81	48.21	912.95
	Adafactor		2.87	1,395.03			54.08	
	small_fc_lopt (naive)		OOM	OOM			OOM	
	small_fc_lopt (CUDA)		1.98	2,025.26			681.78	
	VeLO (naive)		OOM	OOM			OOM	
	VeLO (CUDA)		2.24	1,782.43			443.13	

Table 5. **Per-step training breakdown (averaged over 40 steps).** We compare AdamW, Adafactor, small_fc_lopt, and the VeLO optimizer in both naive (PyTorch) and CUDA-accelerated implementations. CUDA versions significantly reduce optimizer step time and memory overhead, enabling competitive throughput—especially for large models.

H FEATURES FOR LEARNED OPTIMIZER INPUT

In the following section, we introduce the input features for `small_fc_lopt` and `VeLO` in Tables 6 and 7, respectively.

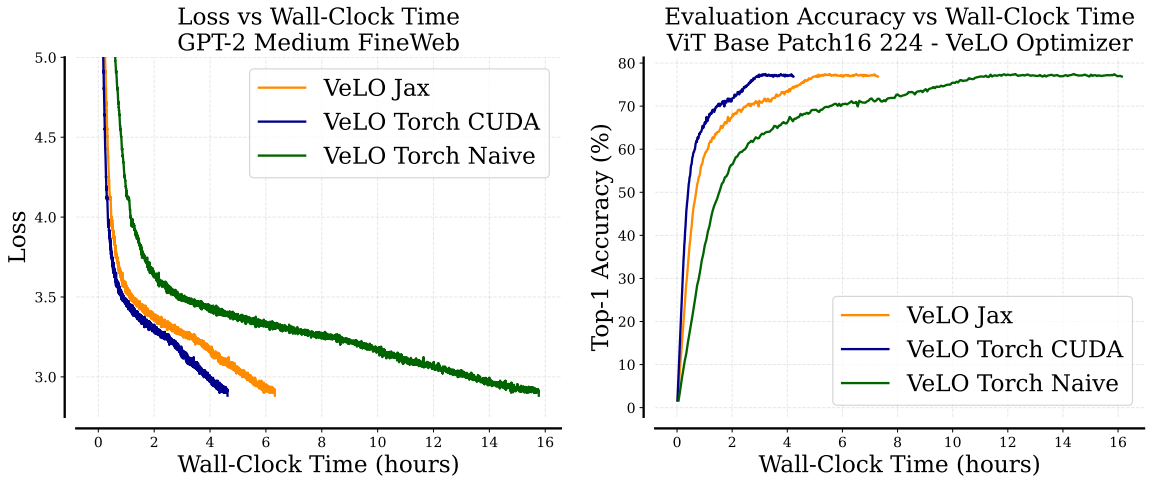
Table 6. **MLP input features for `small_fc_lopt`.** We replicated the table from (**?**) for the reader's convenience with only slight modifications; We also keep the same figure caption. All the coefficients, β_i, are learnable parameters adjusted during meta-optimization. All feature calculations and scalings are reported for a hidden weight matrix $\boldsymbol{W} \in \mathbb{R}^{m \times n}$ in an optimizee network following our proposed μ-parameterization. Here, n is the width and $m = kn$ for some constant $k \in \mathbb{R}$. **Notation.** The table will use $\nabla_{t,i}$ or $\nabla_{t,j}$ to indicate the variable's dependence on time t and coefficient β_i or β_j, respectively. $(\nabla_{t,j})_{r,c}$ will designate indexing into row r and column c of the quantity $\nabla_{t,j}$.

Type	#	Description	Accumulator Update/Equation
Accumulators	3	Momentum accumulators with coefficients $\beta_i, i \in \{1,2,3\}$.	$\boldsymbol{M}_{t,i} = \beta_i \boldsymbol{M}_{t-1,i} + (1-\beta_i)\nabla_t$
	1	Second moment accumulator with coefficient β_4.	$\boldsymbol{V}_t = \beta_4 \boldsymbol{V}_{t-1} + (1-\beta_4)\nabla_t^2$
	3	Adafactor row accumulator with coefficients $\beta_i, i \in \{5,6,7\}$.	$\boldsymbol{r}_{t,i} = \beta_i \boldsymbol{r}_{t-1,i} + (1-\beta_i)\ \texttt{row\_mean}(\nabla_t^2)$
	3	Adafactor accumulator with coefficients $\beta_i, i \in \{5,6,7\}$.	$\boldsymbol{c}_{t,i} = \beta_i \boldsymbol{c}_{t-1,i} + (1-\beta_i)\ \texttt{col\_mean}(\nabla_t^2)$
Accumulator Features	3	Momentum values normalized by the square root of the second moment for $i \in \{5,6,7\}$.	$\frac{\boldsymbol{M}_{t,i}}{\sqrt{\boldsymbol{V}_t}}$
	1	The reciprocal square root of the second moment value.	$\frac{1}{\sqrt{\boldsymbol{V}}}$
	6	The reciprocal square root of the Adafactor accumulators.	$\frac{1}{\sqrt{\boldsymbol{r}_{t,i}}}$ OR $\frac{1}{\sqrt{\boldsymbol{c}_{t,i}}}$
	3	Adafactor gradient features for $i \in \{5,6,7\}$.	$\nabla_t \cdot \sqrt{\frac{\frac{1}{m}\sum_{h=1}^{m}(\boldsymbol{r}_{t,i})_h}{\boldsymbol{r}_{t,i}\boldsymbol{c}_{t,i}^T}}$
	3	Adafactor momentum features for $i,j \in \{(5,1),(6,2),(7,3)\}$.	$\boldsymbol{M}_{t,j} \cdot \sqrt{\frac{\frac{1}{m}\sum_{h=1}^{m}(\boldsymbol{r}_{t,i})_h}{\boldsymbol{r}_{t,i}\boldsymbol{c}_{t,i}^T}}$
Time Features	11	Time features for $x \in \{1,3,10,30,100,300,1000,3000,10^4,3\cdot 10^4,10^5\}$.	$\tanh\left(\frac{t}{x}\right)$
Parameters	1	Parameter value.	$\boldsymbol{W}_t$
	1	Gradient value.	∇_t
Total	39	–	–

Table 7. **MLP input features for `VeLO`.** We replicated the table from (**?**) for the reader's convenience with only slight modifications; We also keep the same figure caption. All feature calculations and scalings are reported for a hidden weight matrix $\boldsymbol{W} \in \mathbb{R}^{m \times n}$ in an optimizee network following our proposed μ-parameterization. Here, n is the width and $m = kn$ for some constant $k \in \mathbb{R}$. **Notation.** The table will use $\nabla_{t,i}$ or $\nabla_{t,j}$ to indicate the variable's dependence on time t and coefficient β_i or β_j, respectively. $(\nabla_{t,j})_{r,c}$ will designate indexing into row r and column c of the quantity $\nabla_{t,j}$.

Type	#	Description	Accumulator Update/Equation
Accumulators	3	Momentum accumulators with coefficients $\beta_i, i \in \{1,2,3\}$.	$\boldsymbol{M}_{t,i} = \beta_i \boldsymbol{M}_{t-1,i} + (1-\beta_i)\nabla_t$
	1	Second moment accumulator with coefficient β_4.	$\boldsymbol{V}_t = \beta_4 \boldsymbol{V}_{t-1} + (1-\beta_4)\nabla_t^2$
	3	Adafactor row accumulator with coefficients $\beta_i, i \in \{5,6,7\}$.	$\boldsymbol{r}_{t,i} = \beta_i \boldsymbol{r}_{t-1,i} + (1-\beta_i)\ \texttt{row\_mean}(\nabla_t^2)$
	3	Adafactor accumulator with coefficients $\beta_i, i \in \{5,6,7\}$.	$\boldsymbol{c}_{t,i} = \beta_i \boldsymbol{c}_{t-1,i} + (1-\beta_i)\ \texttt{col\_mean}(\nabla_t^2)$
Accumulator Features	3	Momentum values normalized by the square root of the second moment for $i \in \{5,6,7\}$.	$\frac{\boldsymbol{M}_{t,i}}{\sqrt{\boldsymbol{V}_t}}$
	1	The reciprocal square root of the second moment value.	$\frac{1}{\sqrt{\boldsymbol{V}}}$
	6	The reciprocal square root of the Adafactor accumulators.	$\frac{1}{\sqrt{\boldsymbol{r}_{t,i}}}$ OR $\frac{1}{\sqrt{\boldsymbol{c}_{t,i}}}$
	3	Adafactor gradient features for $i \in \{5,6,7\}$.	$\nabla_t \cdot \sqrt{\frac{\frac{1}{m}\sum_{h=1}^{m}(\boldsymbol{r}_{t,i})_h}{\boldsymbol{r}_{t,i}\boldsymbol{c}_{t,i}^T}}$
	3	Adafactor momentum features for $i,j \in \{(5,1),(6,2),(7,3)\}$.	$\boldsymbol{M}_{t,j} \cdot \sqrt{\frac{\frac{1}{m}\sum_{h=1}^{m}(\boldsymbol{r}_{t,i})_h}{\boldsymbol{r}_{t,i}\boldsymbol{c}_{t,i}^T}}$
Parameters	1	Parameter value.	$\boldsymbol{W}_t$
	1	Gradient value.	∇_t
	1	Clipped gradient value.	$\texttt{clip}(\nabla_t, -0.1, 0.1)$
Total	29	–	–

I WALL CLOCK TIMES FOR LANGUAGE MODELING AND VISION TRANSFORMERS



(a) GPT-2 Medium FineWeb

(b) ViT/B ImageNet

Figure 12. **Wall-clock training time across different implementations of VeLO for (a) language and (b) vision tasks.** We observe that our CUDA kernel significantly accelerates training, leading to substantially improved wall-clock training times over the Jax and Naive implementations.

J KERNEL NAME DESCRIPTIONS

Table 8. CUDA Kernel Name Legend

Kernel Name	Description
GEMM	General Matrix Multiply (ampere_sgemm), single-precision dense matmul
GEMM_Reduce	Split-K reduction kernel for tiled GEMM accumulation
GEMV	General Matrix-Vector multiply (gemvx::kernel)
GEMV_Batched	Batched GEMV via cuBLAS (cublasGemvParamsEx)
MSE	Mean Squared Error forward kernel
MSE_Backward	Gradient kernel for MSE loss
Mean_Reduce	Reduction kernel computing the mean (MeanOps)
Reduce_Max	Reduction kernel computing the maximum (MaxOps)
Reduce_MaxNaN	Reduction kernel computing the maximum, NaN-aware (MaxNanFunctor)
Reduce_MaxNaN_512	Same as above with 512-thread block configuration
Reduce_Max_128	Max reduction with 128-thread block configuration
Reduce_1Block	Single-block reduction kernel
Add	Elementwise addition
Multiply	Elementwise multiplication
Divide	Elementwise division
Sqrt	Elementwise square root
Pow	Elementwise power
Lerp	Elementwise linear interpolation
Fill	Elementwise fill with constant (FillFunctor)
Elementwise	Generic elementwise kernel (unclassified)
Compute_Squared_Average_LO	Computes squared gradient averages for the learned optimizer
Apply_LO	Applies the learned optimizer parameter update
CatArray	Batched concatenation along a tensor dimension
CatArray_1D	Batched concatenation, 1-D tensors
CatArray_2D	Batched concatenation, 2-D tensors
CatArray_3D	Batched concatenation, 3-D tensors