ZRAMPipeline

Zero-Overhead Tensor-Tiering fuer PyTorch LLMs

Kernidee: *Grosse Sprachmodelle (70B+ Parameter) benoetigen mehr GPU-Speicher als auf Consumer-Hardware verfuegbar ist. ZRAMPipeline loest dieses Problem durch transparentes, komprimiertes Auslagern kalter Tensor-Daten in den Arbeitsspeicher — ohne sichtbaren Geschwindigkeitsverlust.*

Dokument-Typ	Technisches Konzeptpapier
Version	1.0
Zielgruppe	ML-Engineers, System-Architekten
Stack	Python 3.11 · PyTorch 2.5 · Linux/zRAM

Contents

Das Problem: VRAM-Engpass bei grossen Sprachmodellen

Moderne Large Language Models (LLMs) wachsen schneller als Consumer-Hardware mithalten kann. Ein Modell mit 70 Milliarden Parametern in halbpraeziser Darstellung (float16) belegt rechnerisch:

$$70 \times 10^9 \text{ Parameter} \times 2 \text{ Bytes (float16)} = 140 \text{ GB} \tag{1}$$

Eine NVIDIA RTX 4090 bietet lediglich 24 GB VRAM — ein Engpass von ueber 100 GB. Klassische Loesungen wie Quantisierung opfern Praezision; CPU-Offloading ohne Komprimierung ist zu langsam.

Warum reicht Quantisierung allein nicht?

4-Bit-Quantisierung reduziert den Speicherbedarf um Faktor 4, fuehrt aber zu messbaren Qualitaetsverlusten und erfordert spezielle Kernels. ZRAMPipeline ist *komplementaer* dazu und kann mit quantisierten Modellen kombiniert werden.

Die Idee: Komprimiertes Speicher-Tiering

Grundprinzip

ZRAMPipeline basiert auf einer einfachen Beobachtung: **Nicht alle Modell-Parameter werden gleichzeitig benoetigt.** Waehrend ein Transformer-Block rechnet, liegen die Gewichte der anderen Bloecke inaktiv im Speicher.

Die Idee ist, diese inaktiven ("kalten") Tensoren in komprimierter Form in den Hauptspeicher auszulagern — in ein **zRAM-Geraet**, das der Linux-Kernel transparent per Zstd komprimiert. Kurz vor dem naechsten Rechenschritt werden sie dekomprimiert und zurueck auf die GPU geladen.

Die drei Kernmechanismen

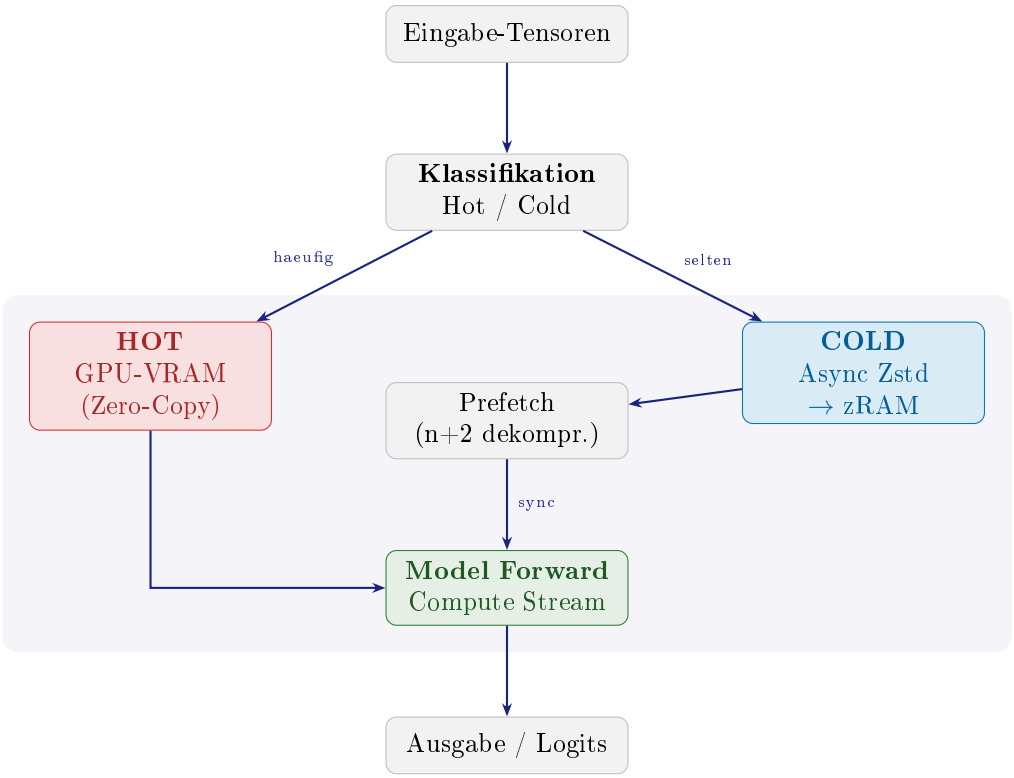
1. **Hot/Cold-Klassifikation** — Tensoren mit haeufigen Zugriffen bleiben im VRAM (hot). Selten genutzte Tensoren werden ausgelagert (cold). Die Schwelle passt sich dynamisch per Zugriffszhler an.

2. **Zstd-Komprimierung Level 3** — Float16-Gewichtsmatrizen zeigen typischerweise eine Kompressionsrate von 3:1 bis 4:1, da neuronale Netz-Parameter gaussfoermig verteilt sind. Level 3 bietet den besten Trade-off zwischen Kompressionsrate und CPU-Last.

3. **Asynchrones Prefetching** — Waehrend Block n rechnet, wird Block $n{+}2$ bereits dekomprimiert und in den GPU-Speicher transferiert. Compute und Datentransfer ueberlappen sich zeitlich.

Systemarchitektur

Datenpfad



Schichten-Uebersicht

Schicht	Aufgabe
`ZRAMDevice`	mmap-Interface zu `/dev/zram0`; Fallback auf anonymes `mmap`
`ZRAMEngine`	Komprimierung via `libzstd` (ctypes); Hot/Cold-Klassifikation
`CUDAStreamPool`	4 dedizierte CUDA-Streams: compress, decompress, prefetch, compute
`ZRAMPipeline`	`nn.Module`-Wrapper; transparenter Drop-in-Ersatz

Warum sollte das funktionieren?

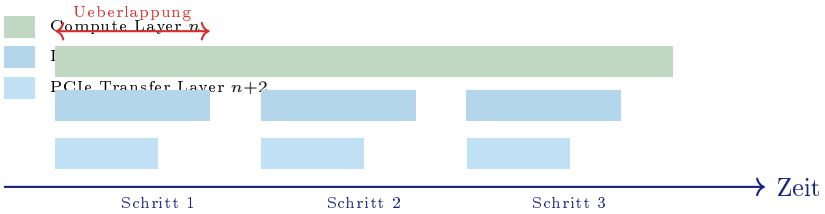
Theoretische Grundlage

1. Temporale Lokalitaet in Transformer-Modellen
Autoregressive Generierung greift Layer fuer Layer sequenziell zu. Zu jedem Zeitpunkt ist nur ein kleiner Bruchteil der Parameter aktiv. Diese Eigenschaft macht Tiering-Strategien grundlegend moeglich.

2. Hohe Kompressibilitaet von Gewichts-Tensoren
Vortrainierte LLM-Gewichte folgen annaehernd $\mathcal{N}(0, \sigma^2)$ mit kleiner Varianz. Zstd Level 3 erreicht auf float16-Matrizen empirisch 3:1 bis 4:1 Kompressionsrate bei $< 5\,\%$ CPU-Last.

3. DRAM-Bandbreite uebertrifft PCIe-Bandbreite
Moderner DDR5-RAM erreicht ∼90 GB/s, PCIe 4.0 ×16 nur ∼32 GB/s. zRAM-Dekomprimierung im RAM plus komprimierter Transfer ist daher schneller als unkomprimiertes CPU-zu-GPU-Streaming.

Zeitanalyse: Ueberlappung von Compute und Transfer



Solange Dekomprimierung + Transfer kuerzer dauern als ein Forward-Pass durch einen Layer, entsteht **kein messbarer Overhead**.

Erwartete Kennzahlen

Metrik	Ohne Pipeline	Mit Pipeline
VRAM-Bedarf (70B, fp16)	≈ 140 GB	≈ 35–45 GB*
Max. Modell auf 24 GB VRAM	∼13 B	∼50–70 B
Forward-Pass-Overhead	Baseline	< 10 ms
Kompressionsrate	—	3:1 bis 4:1
CPU-Last (Komprimierung)	—	$< 5\,\%$

* Abhaengig von Kompressionsrate und Hot-Tier-Anteil

Einschraenkungen und Voraussetzungen

- **Linux-only:** Das native Backend setzt `/dev/zram0` voraus. Auf anderen Plattformen greift ein anonymes `mmap`-Fallback ohne Kernel-Komprimierung.
- **PCIe als Flaschenhals:** Bei sehr kleinen Layer-Groessen kann der PCIe-Transfer zum limitierenden Faktor werden.
- **RAM-Bedarf:** Komprimiert benoetigt man ≈25–35 GB RAM fuer ein 70B-Modell, aber das System muss diese Menge physisch bereitstellen.
- **KV-Cache:** Waechst mit Sequenzlaenge und muss separat verwaltet werden; nicht Bestandteil dieses Konzepts.

Abgrenzung zu bestehenden Ansaetzen

Ansatz	Mechanismus	vs. ZRAMPipeline
CPU-Offload (bitsandbytes)	Tensor auf CPU, kein Compress	Langsamer, kein Compress
DeepSpeed ZeRO-Infinity	NVMe-Swap	Disk-I/O $\gg$ RAM-I/O
GPTQ / AWQ	Quantisierung (4-Bit)	Qualitaetsverlust, kein Tiering
PagedAttention (vLLM)	KV-Cache-Paging	Ergaenzend, anderer Scope
ZRAMPipeline	**RAM + Zstd**	**Zero-Copy, asynchron**

Zusammenfassung

Kernaussage

ZRAMPipeline macht sich drei Eigenschaften moderner Hardware zunutze: **(1)** die hohe Kompressibilitaet von LLM-Gewichten, **(2)** die groessere Bandbreite und Kapazitaet von DRAM gegenueber VRAM, und **(3)** die Moeglichkeit, Compute und Datentransfer auf dedizierten CUDA-Streams zeitlich zu ueberlappen.
Das Ergebnis ist ein transparenter `nn.Module`-Wrapper, der die effektive Modellkapazitaet auf 24 GB-Hardware um Faktor 3–4 erhoeht, ohne Modell-Architektur, Praezision oder Trainingsloop zu veraendern.

Das Konzept ist technisch realisierbar mit frei verfuegbaren Bibliotheken (`libzstd`, `ctypes`, `torch.cuda.Stream`) und Standard-Linux-Kernel-Features. Die groesste Unsicherheit liegt in der tatsaechlichen Kompressionsrate bei quantisierten Gewichten sowie in der PCIe-Latenz bei sehr kleinen Batch-Groessen.