ADAPTIVE WORKING SET STRATEGY FOR TRANSFORMER TRAINING: AUTOMATIC DATA SIZING, OFFLOADING, AND RECOMPUTATION POLICIES

Andrew Sheinberg [1] **Kai Li** [1]

ABSTRACT

The rapid expansion of large language models and long-context training has intensified GPU memory capacity bottlenecks. Existing single-GPU solutions — reduced micro-batch size, offloading, and recomputation — enable greater scale, but often sacrifice throughput due to exposed latency overheads, exposed communication costs, or adding redundant FLOPs. Moreover, these techniques typically require manual tuning based on combination of hardware environment, model size/architecture, and context length. We present Adaptive Working Set Strategy (AdaWS), an orchestration procedure that is general and adaptable to a specific training environment (hardware + model + sequence length). AdaWS maintains a small working set in GPU memory and conducts asynchronous, bidirectional data transfers with secondary host memory. Our generalized computation and communication schedules are conducive to automatic discovery of appropriate runtime decisions based on training environment: micro-batch sizing, offloading policy, and recomputation policy. In turn, our approach offers a portable and user-friendly solution. Through timely prefetching and recomputation avoidance, AdaWS surpasses the throughput of state-of-the-art baselines while using far less GPU memory: for instance, 18 GB vs. 77 GB for Dense-15B and 18 GB vs. 75 GB for Sparse-16B×3B. The system particularly excels at long-context training, where ample computation density allows for full utilization of the CPU-GPU memory complex. AdaWS improves the throughput of long-context training by 1.4×–2.2× and extends trainable context lengths by 4× under typical memory budgets. As a result, our approach enables large-model or long-context training on both datacenter and commodity GPUs at high efficiency. We open source the code at: https://github.com/als244/awsm_dataflow.

1 INTRODUCTION

Modern GPU-accelerated computing systems deliver enormous parallel processing power but are increasingly constrained by limited onboard memory capacity. As model sizes and data footprints continue to grow, GPU memory capacity has emerged as a key bottleneck that restricts both scalability and efficiency.

This limitation has become even more critical with recent shifts toward long-context training and Mixture-of-Experts (MoE) (Jacobs et al., 1991; Shazeer et al., 2017) variants for large language models (LLMs). Training on long sequences, which significantly increases both computation and memory demand, typically requires modifications to usual standard training process. Sparse models, with large numbers of parameters and increased batch sizes, further exacerbate memory issues.

[1]Department of Computer Science, Princeton University, Princeton, NJ, USA. Correspondence to: Andrew Sheinberg <asheinb@princeton.edu>.

Preprint. Work in progress.

To combat increased GPU memory footprint demands, developers are often forced into difficult tradeoffs: reducing batch size, offloading training states, relying on recomputation strategies, or sharding models across multiple devices. These inter-dependent, performance-degrading tradeoffs highlight the urgent need for more intelligent and adaptive GPU memory management solutions.[1]

The working set is a well-known concept introduced in the late 1960s (Denning, 1967) to model program behavior. It refers to the collection of memory pages that a program has referenced within a recent time window. Since most programs exhibit strong temporal and spatial locality, a program's working set is typically much smaller than its total memory footprint. The working set model has been highly effective in guiding virtual memory (VM) management for CPU-based computer systems, where the operating system maintains the active working set in physical memory and uses secondary storage as a backing store for inactive pages.

For Transformer training, the gap between the overall memory footprint and its true working set is striking. As illus-

[1]In this work we focus on single-GPU case.

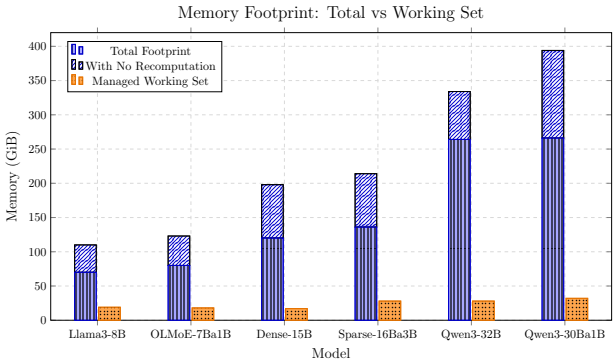


Figure 1. Only a small working set in GPU memory is required for high-throughput training (§4).

trated in Figure 1, only a small fraction of overall application footprint is actively accessed in a fixed window of time. This indicates an opportunity to significantly reduce the GPU memory requirement.

The GPU counterpart of VM is Unified Virtual Memory (UVM), where host memory serves as a backing store for device memory. It is primarily designed to improve programmability (Allen & Ge, 2021). However, when a GPU kernel accesses a page in CPU memory, the GPU is forced to stall; the solution of demand-paging comes with significant overhead and thus is not a practical training solution.

We propose an Adaptive Working Set Strategy (AdaWS) for Transformer training — aimed to minimize the GPU memory requirement while preserving high throughput. We exploit the temporal locality of large AI training to overcome the 30×-100× gap between GPU memory bandwidth and PCIe interconnect bandwidth. We employ efficient prefetching with just-in-time data transfers to maintain a small working set. Our approach ingests information about the hardware environment, model architecture, and sequence length to automatically determine batch sizing, offloading, and recomputation policies. AdaWS then orchestrates a dataflow process that performs timely data transfers between host and GPU memories, entirely overlapped with computation.

AdaWS achieves superior performance and memory efficiency compared to state-of-the-art offloading approaches. The minimal GPU memory required for AdaWS to surpass the throughput of the best baseline systems is dramatically lower (§4): for Llama3-8B, AdaWS requires 56 GB vs. 73 GB; for OLMoE-7B×1B, 20 GB vs. 77 GB; for Dense-15B, 18 GB vs. 77 GB; and for Sparse-16B×3B, 18 GB vs. 75 GB. For long-context sequence lengths that are trainable with existing strategies, AdaWS improves throughput by 1.4× – 2.2×. Moreover, it extends the frontier of trainable context lengths under typical memory budgets by up to 4×, enabling efficient long-context training previously infeasible on the same hardware.

Although we implemented and tested our idea on discrete GPUs, the proposed framework offers a methodology for dynamic data movement in heterogeneous systems. We believe the same principles apply for smaller-scale laptop, mobile, or edge settings with multi-level memory hierarchies atop the main computational processor.

We make the following contributions:

- Enable high-throughput, single-device Transformer training under device and host memory constraints with automated system configuration. (§3)
- Drastically reduce the amount of device memory required for model training via asynchronous, bidirectional dataflow with secondary host memory. (§4.2)
- Enable long-context training with MFU equal to throughput of attention kernel. (§4.4)
- Automatic configuration to adapt to model complexity and GPU characteristics. (§4.6)

2 Background & Related Work

2.1 Transformer Training Pipeline

Transformer models (Vaswani et al., 2017) form the computational backbone of modern large language models (LLMs). Training these models involves executing a sequence of highly structured operations that repeatedly move data through attention, feed-forward, and normalization layers across many tokens and layers. We focus on autoregressive, causal-attention architectures, covering both dense and sparse (Mixture-of-Experts, MoE) variants representative of recent open-source models such as Llama3 (Grattafiori, 2024) and Qwen3 (Yang, 2025).

Memory Composition. The total GPU memory footprint during training can be divided into: (1) model parameters, (2) temporary activations, (3) model gradients, and (4) optimizer state. Among these, activations scale with sequence length × batch size, while gradients and optimizer states scale with the model size Ψ.

2.2 Challenges

Training large Transformers, especially MoE and long-context variants, presents four interrelated challenges:

1. **Minimal GPU Memory.** Model parameters, activations, gradients, and optimizer states often collectively exceed the onboard memory of even high-end GPUs.
2. **High Throughput.** Memory-saving methods often degrade throughput due to recomputation, communication, or data-transfer overheads.

3. **Long-Context Training.** Expanding sequence length normally increases activation footprint, rapidly exhausting GPU memory.

4. **Automatic Configuration.** Manual tuning of batch size, recomputation strategy, offload ratios, and data placement is labor-intensive and hardware-specific.

2.3 Related Work

A variety of approaches have been proposed to address Transformer memory bottlenecks, each targeting a subset of these challenges.

Reducing Batch Size. Shrinking the micro-batch size directly lowers activation memory but typically reduces arithmetic intensity and GPU utilization, harming throughput (addresses challenge (1) but worsens (2)).

Activation Recomputation. Checkpointing or rematerialization (Chen et al., 2016; Jain et al., 2020; Beaumont et al., 2021; Kirisame et al., 2020; Korthikanti et al., 2023) reduces memory by recomputing dropped activations during the backward pass. While effective for challenge (1), it incurs additional compute and limits throughput (2).

Offloading. Systems such as ZeRO-Offload and ZeRO-Infinity (Ren et al., 2021; Rajbhandari et al., 2021), as well as SwapAdvisor and Capuchin (Huang et al., 2020; Peng et al., 2020), move parameters or optimizer states to CPU or NVMe storage. Offloading directly reduces GPU memory pressure (1) but introduces I/O latency that can idle the GPU, lowering throughput (2). These systems also require manual configuration of offload granularity, leaving (4) only partially addressed.

Fused Kernels. Fusing sequential GPU operations (Dao et al., 2022; Hsu et al., 2025) decreases intermediate activations and kernel-launch overheads, improving memory usage (1) and throughput (2), with the price of increased implementation complexity. However, it does not address (3) or (4).

Distributed Training and Sharding. Parallelism strategies—pipeline (Huang et al., 2019; Narayanan et al., 2019), tensor (Narayanan et al., 2021; Korthikanti et al., 2023), expert (Fedus et al., 2022; Dai et al., 2024), and context parallelism (Jacobs et al., 2023; Liu et al., 2024)—expand aggregate memory capacity by distributing model states. Frameworks like ZeRO (Rajbhandari et al., 2020) and FSDP (Zhao et al., 2023) effectively reduce per-GPU memory (1) but at the cost of communication overhead that can limit throughput (2). Context-parallelism strategies target (3), but require expensive hardware and high-bandwidth communication, limiting scalability. Distributed training usually relies on careful manual configuration, failing to address (4).

Parameter-Efficient Fine-Tuning (PEFT). Approaches such as LoRA (Hu et al., 2022) and QLoRA (Dettmers et al., 2023) train only a small subset of parameters, reducing the memory footprint (1) and compute cost. However, they alter the optimization objective and are not applicable to full pretraining or finetuning, leaving (2)–(4) largely unaddressed.

Summary. Overall, prior techniques each mitigate part of the GPU memory bottleneck but fall short of simultaneously addressing all four challenges. In particular, existing solutions either sacrifice throughput, require manual tuning, or cannot extend sequence length effectively.

3 Adaptive Working Set

Our objective is to design a unified framework that addresses all four challenges identified above—minimizing GPU memory usage, achieving high throughput, enabling long-context training, and providing automatic configuration—specifically for Transformer training. We call it **Adaptive Working Set Strategy (AdaWS)**.

AdaWS is based on three key ideas:

- **Generalized Computation and Communication Schedules**: Procedure for ensuring correct dependencies while carrying out asynchronous data transfers. *AdaWS orchestrates just-in-time prefetching to ensure dependencies are available when processor requires them — providing illusion of larger GPU memory capacity without sacrificing performance*.
- **Awareness of Training Environment**: Adapts data placement, batch sizing, offloading, and recomputation decisions based on discovered hardware environment (i.e. memory capacities, interconnect bandwidth, processor speed) in conjunction with model scale and sequence length. These runtime choices parameterize the general template.
- **Integrated Activation Offloading & Recomputation Policies**: Balances benefits of saving activations (minimizing recomputation) with potential interconnect bandwidth bottlenecks. AdaWS employs a Dynamic Programming solver (accurate solution within 100s of μs) to determine the best activations to offload (forward) and subsequently fetch (backward). As a result, our approach fully leverages limited interconnect bandwidth whilst avoiding congestion and communication-induced stalls.

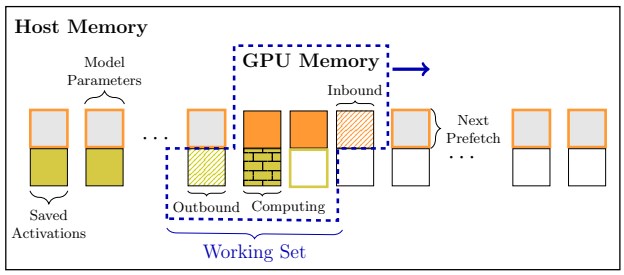
3.1 Sliding Window & Prefetching

AdaWS enables training transformers that exceed GPU memory capacity by maintaining a small, dynamically managed *working set* of layers in device memory at any given time. Rather than loading the entire model into GPU memory, AdaWS retains a *depth-wise slice*—a limited set of

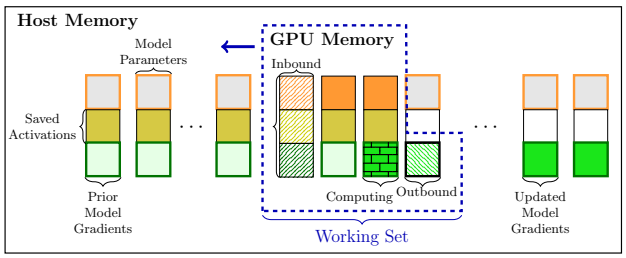
consecutive layers—while using host memory as a backing store. Training can proceed as long as the current layer's dependencies are resident in GPU memory.

After completing a block, AdaWS proactively **prefetches** the data required for upcoming layers while asynchronously evicting data no longer needed. If a dependency is not yet available in GPU memory when required, computation stalls briefly until the transfer completes and data is ready. This *sliding-window* mechanism, coupled with just-in-time prefetching, ensures high utilization by overlapping computation with data movement whilst maintaining a compact memory footprint.

During the forward pass, AdaWS computes activations and transfers the results back to secondary host memory, saving as many as possible without causing idle time (see 3.2). Activations associated with the last |window| layers stay in GPU memory as these will be the next accessed activations (LIFO ordering). After reaching the head and computing the loss, backpropagation begins to compute model gradients.



(a) Sliding Forwards.



(b) Sliding Backwards.

Figure 2. Adaptive Working Set Strategy for DNN Training.

In Figure 2, each column represents a Transformer layer, with all training states and most saved activations stored in the host memory. The dashed blue box indicates the working set in GPU memory. There is a table in GPU memory containing inter-layer transitions (residual streams for forward and backward passes) along with two context-windows (both sized proportional to micro-batch size, though these are not shown in the figure for simplicity). Unlike inference, on device context-windows are reused and reset for each layer.

In Figure 2a, two layers are readily available (orange boxes), with a third currently transferring from the host memory (orange diagonal lines). The GPU computes activations for the layer shown with brick pattern. Activations from the previous layer (left) are marked as outbound, transferring to backing host memory. The yellow-outlined box to the right of the current computation represents an empty buffer that will be used for the next layer's activation workspace. Once the outbound activation transfer completes, we mark the next non-empty activation slot as available (right of the yellow box). After the computation finishes, we replace model parameters with the next layer outside the current window. The window slides forward until reaching the final layer, where it contains the last three layers and activations as readily-available dependencies for backpropagation.

Figure 2b shows the working set during backpropagation, with the sliding window moving in reverse. Computing each backward layer requires model weights, forward activations, and prior gradients (for multiple rounds of gradient accumulation) as input. Each backward layer updates gradients before sending updated results back to the host memory. After completing the current backward layer, we prefetch dependencies (weights, activations, gradients) for next downstream layer not in our window.

The general pseudocode forward and backward passes can be found in Appendix A.

This sliding window approach with asynchronous data transfers enables training large models with minimal GPU memory. However, achieving high throughput requires awareness of training environment to avoid stalls and minimize recomputation.

3.2 Avoiding Stalls & Recomputation

3.2.1 Overview

The key to achieving high throughput is to avoid GPU idle time and additional computation. We must balance these interrelated aspects when determining appropriate data sizing, window sizing, and saved activation policy. For a given layer (i.e. a column in Figure 2), the size of weights/gradients is fixed, but the amount of memory for activations is proportional to micro-batch size. AdaWS first determines an appropriate micro-batch size, which then gives a concrete size for each complete layer, allowing us set the window size in terms of # of layers. Then, we set saved activation policy to minimize recomputation while avoiding stalls. These adaptive, runtime decisions are tightly integrated with hardware environment, model architecture, and sequence length. We assume a generalized Transformer architecture parameterized by the table in Figure 10b.

Potential Stalls. Stalls can occur in three situations:

1. Weight dependencies are not ready during the forward pass,
2. Activation write-buffer is not available during the forward pass,
3. Weights, forward activations, or gradient dependencies are not ready during the backward pass.

3.2.2 Data Sizing

How much data should we process during each gradient accumulation round?

Forward Requirements

To address (1) we must decide how much work (micro-batch size) to supply per forward-backward round. As long as a forward layer's computation exceeds the weight prefetch duration, the sliding window contents will be ready when needed. This is a conservative lower-bound[2], but provides strong guarantees. Given a model architecture with ψ_{active} active parameters per layer, X sequences of length S using full causal attention where $d_{\text{attn}} = n_Q \cdot h_{\text{dim}}$, practical processing speed of P FLOPs/sec, we can accurately estimate the forward computation will take[3]:

$$T_{\text{fwd}} = \frac{2 \cdot X \cdot S \cdot (\psi_{\text{active}} + S \cdot d_{\text{attn}})}{P} \text{ sec}$$

We also know the duration it takes to transfer weights: assume ψ_{total} total parameters per layer with average datatype size of $|\tau_{\text{param}}|$ bytes and interconnect bandwidth of BW_{inter} bytes/sec. The weight transfer time is:

$$T_{\text{param}} = \frac{|\tau_{\text{param}}| \cdot \psi_{\text{total}}}{BW_{\text{inter}}} \text{ sec}$$

Now we wish to satisfy (1) which conservatively implies $T_{\text{fwd}} > T_{\text{param}}$, so we solve for an initial X as

$$\#\text{ Sequences} = X \geq \frac{|\tau_{\text{param}}| \cdot \psi_{\text{total}} \cdot P}{2 \cdot S \cdot (\psi_{\text{active}} + S \cdot d_{\text{attn}}) \cdot BW_{\text{inter}}}$$

though, we will correct this to account for gradient transfers.

We determine the realistic hardware constants P and BW_{inter} at runtime.

Backward Requirements

To address (3) we assume that our saved activation policy (§3.2.4) ensured no idle time occurred during forward pass and thus already satisfied (2). With this assumption we have a bound on the quantity of activations that will fetched during a window-sized period of backwards layers (each layer may save/fetch a different quantity, but the total bytes is bounded for any span of consecutive window-sized layers). If our final configuration ends up setting a window size of W layers, satisfying (2) guarantees the maximum number of bytes saved is bounded by:

$$\text{Act Bytes Saved In Full Period} \leq \frac{W \cdot T_{\text{fwd}}}{BW_{\text{inter}}}$$

In turn, for any window it will take $\leq W \cdot T_{\text{fwd}}$ sec to fetch these back.

Additionally, gradients might be a different datatype with average datatype size $|\tau_{\text{grad}}|$ where

$$T_{\text{grad}} = \frac{|\tau_{\text{grad}}| \cdot \psi_{\text{total}}}{BW_{\text{inter}}} = \frac{|\tau_{\text{grad}}|}{|\tau_{\text{param}}|} \cdot T_{\text{param}} \text{ sec}$$

Thus during any period of W backward layers the total inbound transfers (activations + weights + gradients) is bounded by:

$$\text{Bwd Window Transfer} \leq W \cdot \left(\left(1 + \frac{|\tau_{\text{grad}}|}{|\tau_{\text{param}}|} \right) \cdot T_{\text{param}} + T_{\text{fwd}} \right) \text{ sec}$$

We must ensure sufficient micro-batch size such that computing W backward layers $>$ Bwd Window Transfer. A lower bound for processing time of backwards with full parameter training is:

$$T_{\text{bwd}} \geq 2 \times T_{\text{fwd}}$$

where the realistic computation time may be larger with recomputation or long sequences (we use Flash Attention (Dao et al., 2022) which has factor of 5 instead of 4 for the amount of bwd attention FLOPs). Thus for a window of W layers it will take $\geq 2 \cdot W \cdot T_{\text{fwd}}$ sec. Now we can further refine $\#$ Sequences to ensure that $W \cdot T_{\text{bwd}} \geq$ Bwd Window Transfer, yielding the inequality:

$$2 \cdot W \cdot T_{\text{fwd}} \geq W \cdot \left(\left(1 + \frac{|\tau_{\text{grad}}|}{|\tau_{\text{param}}|} \right) \cdot T_{\text{param}} + T_{\text{fwd}} \right)$$

This provides a relationship between T_{fwd} and T_{param} to ensure the total computation time during any complete backwards window exceeds the total transfer time. Because the backwards pass starts with a fully populated window, by induction we are guaranteed that every additional layer (in reverse) will be ready.

[2] With a sufficiently wide window relative to model depth, prefetch time can actually exceed computation time without causing idleness as consequence of initial headstart, though this analysis becomes complex.

[3] Our codebase supports variable length sequences, but here we assume fixed sequence length for simplicity.

We now obtain:

$$T_{\text{fwd}} \geq \left(1 + \frac{|\tau_{\text{grad}}|}{|\tau_{\text{param}}|}\right) \cdot T_{\text{param}}$$

as a stricter bound than (1) initially implied. Given this relationship we determine an appropriate micro-batch size as

$$\boxed{\#\text{ Sequences} = X \geq \frac{(|\tau_{\text{param}}| + |\tau_{\text{grad}}|) \cdot \psi_{\text{total}} \cdot P}{2 \cdot S \cdot (\psi_{\text{active}} + S \cdot d_{\text{attn}}) \cdot BW_{\text{inter}}}}$$

This micro-batch size is compatible with the general AdaWS schedule (§3.1 and Appendix A) and likely does not apply to other offloading/recomputation strategies.[4] The number of gradient accumulation rounds is then set based upon user-inputted global batch size.

3.2.3 *Window Sizing*

How many layers to place in GPU memory?

Once we determine the amount of data to process each gradient accumulation round we have a wholistic picture of the total bytes required per complete layer. The total activation size per layer is:

$$A_{\text{bytes}} = |\tau_{\text{act}}| \cdot X \cdot S \cdot (d_{\text{attn}} + 2 \cdot d_{\text{ctx}} + d_{\text{model}} + 2 \cdot (E_{\text{shared}} + k) \cdot d_{\text{expert}})$$

Additionally we add in the fixed size of parameters and gradients to obtain a full layer size of:

$$L_{\text{bytes}} = (|\tau_{\text{param}}| + |\tau_{\text{grad}}|) \cdot \psi_{\text{total}} + A_{\text{bytes}}$$

Further, we account for static buffer space: full training state of embedding layer, unembedding layer, transition table ($|\tau_{\text{act}}| \cdot X \cdot S \cdot d_{\text{model}}$), context windows ($|\tau_{\text{act}}| \cdot 4 \cdot X \cdot S \cdot d_{\text{ctx}}$), and additional kernel workspace. Let this fixed memory overhead be B total baseline bytes and assume a GPU memory budget of C_{dev}. Finally, we set the window size as:

$$\boxed{W = \frac{C_{\text{dev}} - B}{L_{\text{bytes}}}}$$

3.2.4 *Saved Activation Policy*

How to decide what activations should be offloaded vs. recomputed?

Here we prevent potential stall (2). Our objective to maximize the amount of saved computation while ensuring that activation buffers are available for use in the next period. Given a window size of W we want to satisfy

$$t_{\text{finishTransfer}_i} < t_{\text{startCompute}_{i+W}}$$

to ensure write-buffer is ready. We set O discrete options for possible saved activation choices per layer (in our work we choose $O = 4$, described in Figure **??**). At a minimum we require the input to layer along with KV activations to be saved. We can determine when the start time will occur for all layers based on $t_{\text{startCompute}_i} = i \cdot T_{\text{fwd}}$. Each of the 4 saved activation options per layer has an associated transfer time and optimization value (computation time associated with that level). We pass the vector of full layer compute times (length L for number of layers) along with matrices of transfer times and optimization values (each are $L \times O$) to a DP solver that maximizes value while satisfying the write-buffer ready constraint. [5] The result is a vector of L saved activation choices, where we only consider the first $L - W$ options (the last W are saved on device) and each option is $\in [0, O)$.

4 Evaluation

We evaluate how effectively the proposed **AdaWS** approach addresses the four key challenges identified in §2: minimizing GPU memory usage, achieving high throughput, enabling long-context training, and automatic configuration. We would like to answer the following questions:

- How well does AdaWS reduce GPU memory requirements while achieving high throughput?
- How does AdaWS compare to Nvidia's UVM?
- Can AdaWS support long-context training without performance degradation?
- What are the effects of host memory capacity on AdaWS's performance and scalability?
- How well does AdaWS work with larger models and smaller GPUs?

4.1 Experimental Setup

AdaWS Implementation. Our prototype consists of ≈10K lines of C. We rely on cublasLt for matrix multiplication and FA3 (Shah et al., 2025)/FA2 (Dao, 2023) for attention. All other kernels are implemented in CUDA C for simple integration with our system. We manually manage

[4] We set additional guards on top of basic equation: process at least 1 sequence per round, cap the tokens per round to ensure $W \geq 2$ (§3.2.3), and cap the tokens per round to guarantee sufficient host memory capacity to fit required minimally saved activations.

[5] We implement the Transmission Scheduling DP solver as a standalone library. This module is heavily optimized – with assistance of Claude, Gemini, and Codex – and takes on order of 100-500 μs.

all memory and rely on the GPU runtime for setting dependencies between streams. We found it challenging to apply, debug, and profile our logic in existing frameworks due to nested memory managers and autograd engines, motivating our custom stack. However, we are actively working on integrating with community ecosystems.

Hardware Platform. We evaluate on two environments: (1) an H100 with 512 GiB host memory (≈300 GB/s BW) and (2) an RTX5090 with 192 GiB host memory (≈80 GB/s BW). Both use PCIe 5.0 (≈64 GB/s). These two configurations allow us to evaluate AdaWS's scalability under both server-class and consumer-grade conditions.

Models Evaluated. To cover a spectrum of model scales and activation characteristics, we evaluate three dense and three sparse transformer models 1, all using autoregressive causal attention and standard AdamW (Loshchilov & Hutter, 2019) optimization. The model dimensions are listed in Table 1.

All models use a standard sequence length of 8K tokens unless otherwise specified. Global batch sizes are selected to achieve approximately 1–2% optimizer overhead: 72 sequences for dense models and 180 for sparse models (doubled to 144 and 360, respectively, for DeepSpeed baselines). Throughput is measured at the third steady-state training step.

Baselines and their limitations. We compare AdaWS against strong, state-of-the-art baselines implemented in PyTorch using optimized kernels such as FlashAttention-3 for attention, Quack/Liger (Hsu et al., 2025; qua) for fused linear operations, and ScatterMoE (Tan et al., 2024) for expert processing. Our main baseline systems are the DeepSpeed ZeRO family (Ren et al., 2021; Rajbhandari et al., 2021), including **ZeRO-1/2:** which offloads optimizer state to host memory, and **ZeRO-3:** which offloads parameter and optimizer state.

For dense models, we also include NVIDIA's Unified Virtual Memory (UVM) as a system-level offloading baseline that automatically migrates data between GPU and host memory.

AdaWS is evaluated using a pure 16-bit training state (8Ψ total memory), while standard DeepSpeed configurations maintain FP32 master weights, gradients, and optimizer states (16Ψ total). We retain this configuration since DeepSpeed's pure 16-bit mode (available only in ZeRO-2) exhibited poor stability and subpar throughput. To ensure fairness, we double the global batch size for all DeepSpeed baselines to equalize the ratio of computation to optimizer-step overhead. We apply five levels of layer-wise selective activation checkpointing on top of each ZeRO configuration.

Measuring Throughput. We record MFU based on Peak TFLOPS of 989 for H100 and 209.5 for RTX 5090. We consider the model cost of causal attention with no recompuation. FlashAttention has a factor of 7 for the attention term, but we only count this towards HFU. Let Ψ_{active} be the total number of active, non-embedding parameters. The number of FLOPs per sequence:

$$6S \cdot (\Psi_{\text{active}} + S \cdot L \cdot n_{\text{Q}} \cdot h_{\text{dim}})$$

4.2 GPU Memory Footprint vs. Throughput

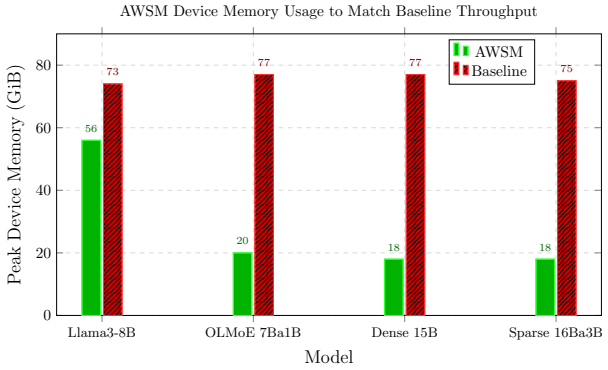


Figure 3. Minimal device memory usage for AdaWS to surpass throughput of best baseline.

To answer the question about how well AdaWS performs in terms of GPU memory requirements while achieving high throughput, we evaluate the impact of GPU memory footprints on end-to-end training of all six models, comparing the AdaWS prototype to the baselines.

Figure 5 shows the footprints of GPU memory and the training performance of 4 smaller models (Llama3-8B, Dense-15B, OLMoE-7Ba1B, and Sparse-16Ba3B). The results show that AdaWS shifts the frontier up and to the left across for all configurations.

AdaWS has substantial performance gains for larger or sparse models. The reason is primarily due to less recomputation, portrayed in Figure 4. Larger models, such as in 5b generate more activations. In this case, the benefits of our scheme become more apparent. The baselines do not save activations in host memory. In order to successfully run, without GPU out-of-memory errors, the baselines must recompute 75-100% of all forward pass computations. Because AdaWS shuffles data in and out, in a timely fashion, we can both keep a small footprint and save + fetch this activation data asynchronously rather than recompute.

Figure 3 shows a summary of the results, depicting the GPU memory level where the throughput of AdaWS surpasses the throughput of highest performing baseline. We see 4x GPU memory reduction for the large or sparse model cases.

Table 1. Characteristics of tested models.

Model	V	L	d_{model}	h_{dim}	n_Q	n_{ctx}	d_{expert}	E_{shared}	E_{routed}	k
Llama3-8B	128256	32	4096	128	32	8	14336	1	0	0
OLMoE-7Bx1B	50304	16	2048	128	16	16	1024	0	64	8
Dense 15B	128256	64	4096	128	32	8	14336	1	0	0
Sparse-16Bx3B	128256	32	3072	128	24	4	768	0	64	8
Qwen3-32B	152064	64	5120	128	40	8	27648	1	0	0
Qwen3-30Bx3B	151936	48	2048	128	32	4	768	0	128	8

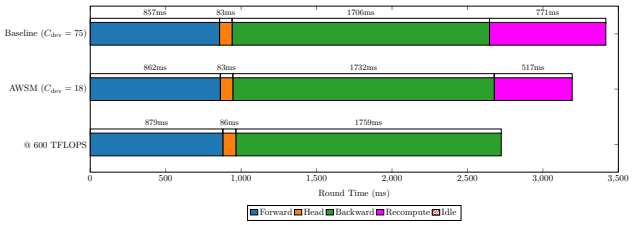


Figure 4. Dense 15B. AdaWS @ 18 GB vs. Baseline @ 75 GB

4.3 Comparison with UVM

Figure 6 presents the results of training three dense models—Llama3-8B, Dense-15B, and Qwen3-32B. Across all configurations, AdaWS achieves substantially higher MFU than UVM with 100%, 50%, and 0% recompute configurations.

The key reason is that UVM relies on a page-fault–driven mechanism, which introduces stalls and demand-paging latency, resulting in unstable performance for large models or long sequence lengths. The massive performance degradation coincides with increased overall application footprint. With a large gap between total footprint vs. GPU memory capacity, UVM severely struggles.

In contrast, AdaWS's proactive management of a small working set through just-in-time prefetching eliminates these stalls, delivering significantly more stable and efficient utilization.

4.4 Long-Context

Following prior work, we adopt Llama3-8B as the reference model to evaluate the effectiveness of long-context training using AdaWS. Table 2 compares AdaWS with two single-device long-context training methods: Mini-Sequence Transformer (MST) (Luo et al., 2025) and Arctic Long Sequence Training (ALST) (Bekman et al., 2025).

For sequence lengths that are trained using existing strategies, AdaWS improves throughput by 1.4×–2.2×. More importantly, AdaWS extends the trainable context length frontier by up to 4× under typical GPU memory budgets. Even under tighter device memory constraints, AdaWS maintains high efficiency with minimal degradation. For example,

	Sequence Length				
Strategy	**32K**	**64K**	**128K**	**256K**	**512K**
MST	37.2% (5.2K)	✗	✗	✗	✗
ALST	24.4% (3.4K)	26.1% (2.7K)	32.4% (2.1K)	✗	✗
AdaWS	57.1% (8.0K)	56% (5.7K)	53.1% (3.6K)	51.9% (2.1K)	42.5% (0.9K)

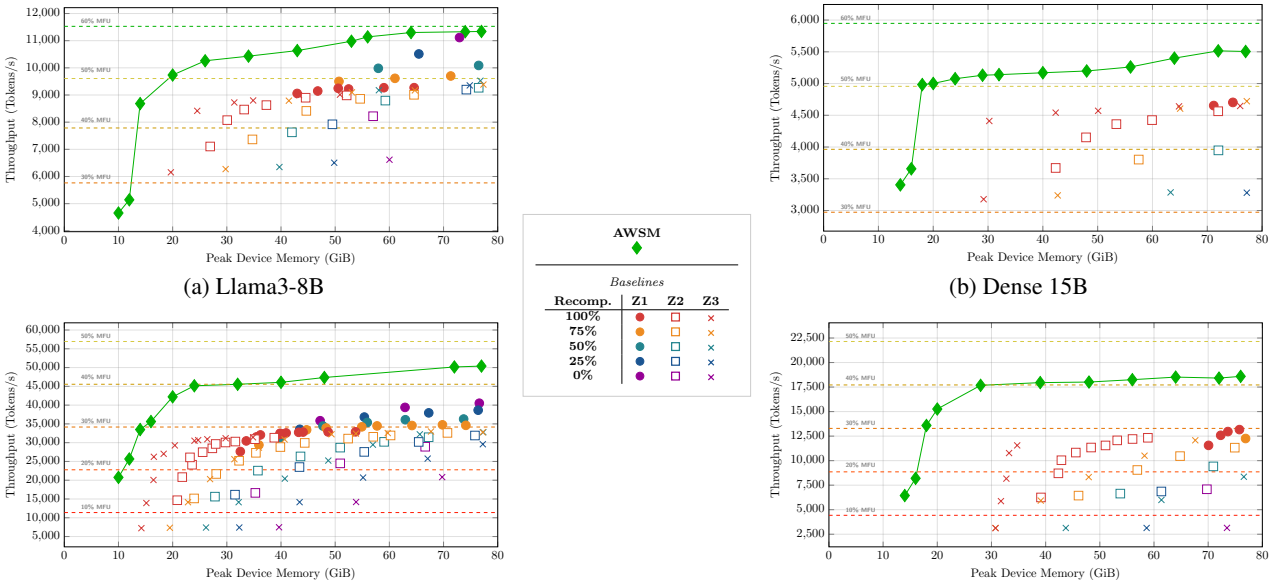
Table 2. MFU (Tok/sec) at Long-Contexts for full BF16 Llama3-8B model on an H100 GPU. The memory budget is $C_{\text{dev}} = 80$ GiB and $C_{\text{host}} = 256$ GiB.

with a device memory capacity of $C_{\text{dev}} = 24$ GiB and sequence length $S = 256$K, AdaWS sustains 2024 Tok/s compared to 2133 Tok/s at full memory utilization (51.4% vs. 51.9% MFU).

At longer sequence lengths, the quadratic cost of attention begins to dominate overall computation. For Llama3-8B, attention operations account for roughly 82% of total FLOPs at $S = 256$K and 90% at $S = 512$K.

As an increasing proportion of time is spent on token-to-token communication rather than forward progression, the system gains sufficient slack to transfer full activations back to host memory without congestion—assuming adequate host capacity. This enables AdaWS to avoid recomputation while maintaining smooth dataflow. Our design assumes that, at minimum, inputs to each Transformer block are retained, making the capacity of the secondary memory pool the dominant factor for long-context scalability.

As illustrated in Figure 7, with sufficiently large secondary memory, AdaWS keeps the GPU consistently supplied with productive work. The resulting MFU for long-context training is nearly identical to the throughput achieved by the optimized FlashAttention-3 kernel. This indicates that AdaWS's automatic working set management is operating near its theoretical limit—there is virtually no remaining headroom for throughput improvement. While AdaWS achieves near-perfect hardware utilization, the inherent computational cost of long-context attention remains an open challenge.



(a) Llama3-8B (b) Dense 15B

(c) OLMoE 7Ba1B (d) Sparse 16Ba3B

Figure 5. Device Memory vs. Throughput across model configurations. Sequence length 8192 on H100.

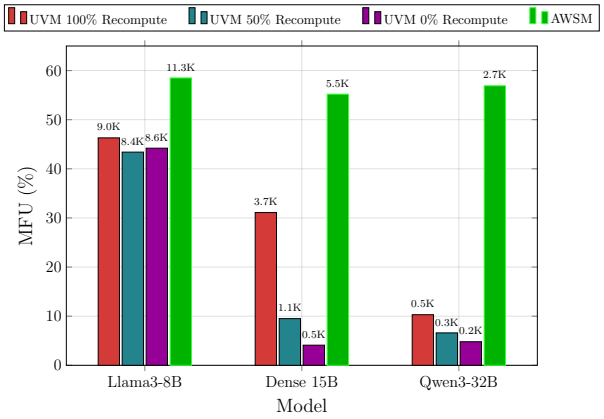


Figure 6. Throughput Comparison with UVM

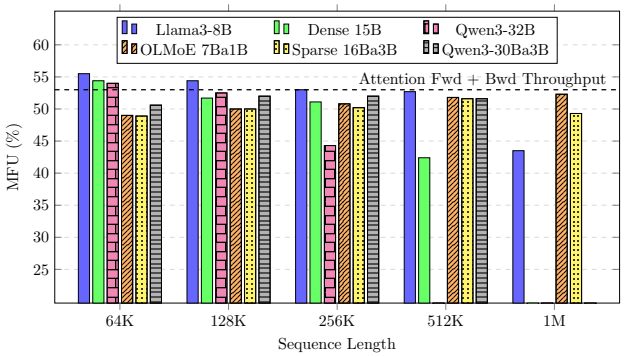


Figure 7. Long-Context on H100 with large host pool 480 GiB. Given sufficient backing memory, MFU approaches the throughput of attention kernel.

4.5 Usage of Host Memory Capacity

Since AdaWS and other memory-optimization approaches rely on host memory as a backing store to reduce GPU memory pressure, it is essential to understand how the host memory requirements and their impacts on performance and scalability.

We monitor the Linux-reported Resident Set Size (RSS) for the experiments described in §4.2, and plot the peak host memory usage in Figure 8. Although we expect the baselines to consume roughly twice as much host memory—due to their use of higher-precision training states (as discussed in §4.1)—the observed footprints are significantly larger.

In particular, ZeRO-3 baselines exhibit host memory consumption between 28Ψ and 32Ψ, nearly double the anticipated training-state size. This excessive usage arises from the nested design of the DeepSpeed engine and the interaction between PyTorch's garbage collection and memory allocator, making it difficult to track and constrain.

In contrast, AdaWS maintains precise control over host buffers. Its host memory footprint corresponds closely to 8Ψ—the expected size of the training state—plus a predictable, small amount of additional space for saved activations. This controlled and transparent memory behavior ensures scalability across varying host capacities and prevents unpredictable over-allocation.

Maintaining a tight host memory footprint is critical for scaling model training, especially in local training and fine-

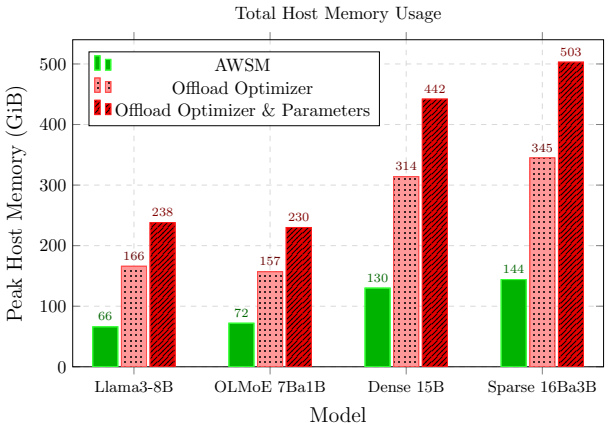


Figure 8. Host Memory Usage

tuning scenarios where host capacity is limited. Figure 9 illustrates that AdaWS enables high-performance training of models that the baselines fail to execute.

In baseline systems, the pure 16-bit precision mode frequently triggers device out-of-memory (OOM) errors, underscoring the importance of coordinated device and host buffer management.

The results for Qwen30B×3B in Figure 9a use a sequence length of 4K (others use 8K) to align with NVIDIA's NeMo benchmark on an 8-GPU DGX-H100 cluster employing expert parallelism.

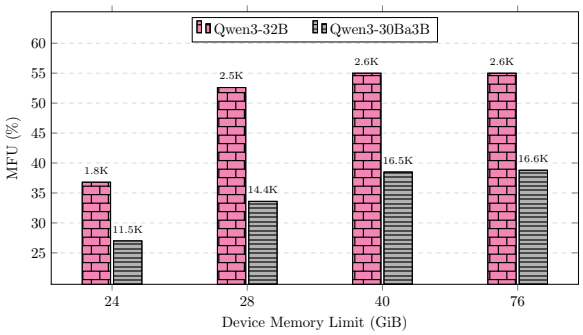
Under comparable conditions, AdaWS achieves 38% higher per-GPU throughput while using only 40 GiB of device memory. This improvement highlights the advantage of localized computation—our single-device setup avoids the collective communication overheads that introduce blocking behavior in multi-GPU pipelines.

Beyond supporting larger models, AdaWS's host buffer management is equally essential for sustaining long-sequence training at near-optimal throughput, achieving better scalability and efficiency.

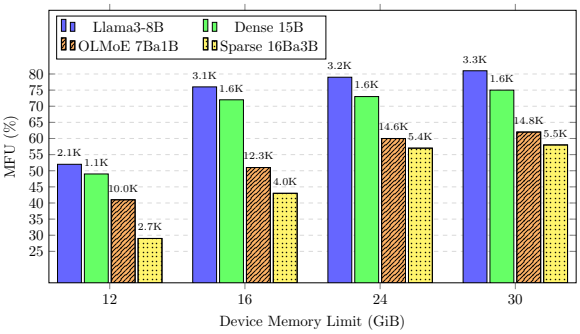
4.6 Large Models and Small GPU

Evaluating the effectiveness of AdaWS's automatic configuration presents a unique challenge because existing baselines lack comparable self-tuning capabilities. Rather than exhaustively testing many combinations of models and hardware configurations, we focus on two representative scenarios where baseline systems fail to work: training larger-scale models and training on a workstation-class GPU with 32GB memory.

Large-Model Evaluation. We test AdaWS on two large-scale Transformer models that cannot be trained under any baseline configurations due to out-of-memory failures. Figure 9a shows that AdaWS successfully adapts to both models without manual tuning and achieves up to **55% MFU**, demonstrating that its automatic configurations and buffer management scale effectively to high-parameter regimes.



(a) H100 Large Model Performance, Host Limit = 320 GB



(b) RTX 5090 Training Performance, Host Limit = 140 GB

Figure 9. Our precise buffer management enables high-throughput large model training, even with low GPU memory.

Workstation-Class GPU We further evaluate AdaWS on workstation grade Nvida RTX 5090 GPU with **12 GB 16 GB**, **24 GB**, and **30 GB** of GPU memory setups. Figure 9b shows that across four smaller models, AdaWS achieves **55% to 80% MFU**. These results confirm AdaWS's ability to maintain high utilization and stable performance even when operating under tight memory budgets.

Overall, these experiments show the robustness of AdaWS across scales: it can train models that exceed the capacity of baseline systems while preserving strong efficiency on smaller, resource-constrained hardware.

5 Limitations and Future Work

AdaWS is developed and evaluated specifically for Transformer architectures, where the separation between model layers and activation dependencies naturally aligns with our working-set abstraction. However, the underlying ideas—dynamic working-set tracking, just-in-time prefetching, and automatic configuration—are more general. We believe similar principles could be applied to other deep learning architectures.

This work focuses on single-GPU training to isolate the effects of automatic memory management without interference from inter-GPU synchronization or communication. We view AdaWS as a *building block* for distributed training systems, where inter-device dependencies, timing coordination, and network bandwidth contention will introduce additional complexity. Extending AdaWS to multi-GPU environments is an important direction for future work.

Our comparisons with baseline systems have practical limitations. Not all frameworks support identical datatype configurations or kernel implementations, which may influence absolute throughput. We have attempted to align settings wherever possible. Despite differences, AdaWS consistently demonstrates superior efficiency across configurations.

All experiments in this paper are conducted on NVIDIA GPUs. Because AdaWS does not depend on hardware-supported paging mechanisms, its design should be portable to other accelerators, such as AMD GPUs, TPUs, or emerging AI chips with hierarchical memory. Nevertheless, validating AdaWS on these platforms will require additional engineering and evaluation, which we leave as future work.

6 Conclusion

This paper presents **Adaptive Working Set (AdaWS)**, a framework that addresses the four major challenges in GPU-based Transformer training: minimizing GPU memory usage, achieving high throughput, enabling long-context training, and providing automatic configuration. AdaWS dynamically manages the active working set through just-in-time prefetching and precise host–device coordination, allowing models far exceeding GPU memory capacity to train efficiently on a single device.

Our experiments demonstrate that AdaWS significantly outperforms state-of-the-art baselines across model sizes and configurations. It achieves up to **4× lower GPU memory usage** while sustaining equal or higher throughput, and extends the frontier of trainable context lengths by **up to 4×**.

In addition to reducing GPU memory, AdaWS also minimizes **host memory consumption**, achieving predictable and tightly bounded usage compared to baseline systems such as ZeRO. This improvement has practical implications for GPU cluster design and scheduling, as it reduces host resource contention and improves training scalability.

This study demonstrates that careful working-set management can mitigate physical GPU memory limitations, achieving efficient, high-throughput, and scalable training for large models.

References

Dao-AILab/quack: A Quirky Assortment of CuTe Kernels. URL https://github.com/Dao-AILab/quack?tab=readme-ov-file.

Allen, T. and Ge, R. In-depth analyses of unified virtual memory system for gpu accelerated computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384421. doi: 10.1145/3458817.3480855. URL https://doi.org/10.1145/3458817.3480855.

Beaumont, O., Eyraud-Dubois, L., and Shilova, A. Efficient combination of rematerialization and offloading for training DNNs. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, NIPS '21, pp. 23844–23857, Red Hook, NY, USA, December 2021. Curran Associates Inc. ISBN 978-1-7138-4539-3.

Bekman, S., Rajbhandari, S., Wyatt, M., Rasley, J., Ruwase, T., Yao, Z., Qiao, A., and He, Y. Arctic Long Sequence Training: Scalable And Efficient Training For Multi-Million Token Sequences, June 2025. URL http://arxiv.org/abs/2506.13996. arXiv:2506.13996 [cs].

Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost, 2016. URL https://arxiv.org/abs/1604.06174.

Dai, D., Deng, C., Zhao, C., Xu, R. X., Gao, H., Chen, D., Li, J., Zeng, W., Yu, X., Wu, Y., Xie, Z., Li, Y. K., Huang, P., Luo, F., Ruan, C., Sui, Z., and Liang, W. DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models, January 2024. URL http://arxiv.org/abs/2401.06066. arXiv:2401.06066 [cs].

Dao, T. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning, July 2023. URL http://arxiv.org/abs/2307.08691. arXiv:2307.08691 [cs].

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, pp. 16344–16359, Red Hook, NY, USA, November 2022. Curran Associates Inc. ISBN 978-1-7138-7108-8.

Denning, P. J. The working set model for program behavior. In *Proceedings of the first ACM symposium*

on Operating System Principles, SOSP '67, pp. 15.1–15.12, New York, NY, USA, January 1967. Association for Computing Machinery. ISBN 978-1-4503-7370-8. doi: 10.1145/800001.811670. URL https://dl.acm.org/doi/10.1145/800001.811670.

Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. QLORA: efficient finetuning of quantized LLMs. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, pp. 10088–10115, Red Hook, NY, USA, December 2023. Curran Associates Inc.

Fedus, W., Zoph, B., and Shazeer, N. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022. URL http://jmlr.org/papers/v23/21-0998.html.

Grattafiori, A. e. a. The Llama 3 Herd of Models, November 2024. URL http://arxiv.org/abs/2407.21783. arXiv:2407.21783 [cs].

Hsu, P.-L., Dai, Y., Kothapalli, V., Song, Q., Tang, S., Zhu, S., Shimizu, S., Sahni, S., Ning, H., Chen, Y., and Wang, Z. Liger-Kernel: Efficient Triton Kernels for LLM Training. July 2025. URL https://openreview.net/forum?id=36SjAIT42G.

Hu, E., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. LORA: LOW-RANK ADAPTATION OF LARGE LAN- GUAGE MODELS. 2022.

Huang, C.-C., Jin, G., and Li, J. SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pp. 1341–1355, New York, NY, USA, March 2020. Association for Computing Machinery. ISBN 978-1-4503-7102-5. doi: 10.1145/3373376.3378530. URL https://dl.acm.org/doi/10.1145/3373376.3378530.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. GPipe: efficient training of giant neural networks using pipeline parallelism. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, number 10, pp. 103–112. Curran Associates Inc., Red Hook, NY, USA, December 2019.

Jacobs, R. A., Jordan, M. I., Nowlan, S. J., and Hinton, G. E. Adaptive Mixtures of Local Experts. *Neural Computation*, 3(1):79–87, March 1991. ISSN 0899-7667. doi: 10.1162/neco.1991.3.1.79. URL https://ieeexplore.ieee.org/document/6797059.

Jacobs, S. A., Tanaka, M., Zhang, C., Zhang, M., Song, S. L., Rajbhandari, S., and He, Y. DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models. *CoRR*, abs/2309.14509, 2023. doi: 10.48550/ARXIV.2309.14509. URL https://doi.org/10.48550/arXiv.2309.14509. arXiv: 2309.14509.

Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Gonzalez, J., Keutzer, K., and Stoica, I. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In Dhillon, I., Papailiopoulos, D., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems*, volume 2, pp. 497–511, 2020. URL https://proceedings.mlsys.org/paper_files/paper/2020/file/0b816ae8f06f8dd3543dc3d9ef196cab-Paper.pdf.

Kirisame, M., Lyubomirsky, S., Haan, A., Brennan, J., He, M., Roesch, J., Chen, T., and Tatlock, Z. Dynamic Tensor Rematerialization. October 2020. URL https://openreview.net/forum?id=Vfs_2RnOD0H.

Korthikanti, V. A., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., and Catanzaro, B. Reducing Activation Recomputation in Large Transformer Models. In Song, D., Carbin, M., and Chen, T. (eds.), *Proceedings of Machine Learning and Systems*, volume 5, pp. 341–353. Curan, 2023. URL https://proceedings.mlsys.org/paper_files/paper/2023/file/80083951326cf5b35e5100260d64ed81-Paper-mlsys2023.pdf.

Liu, H., Zaharia, M., and Abbeel, P. RingAttention with Blockwise Transformers for Near-Infinite Context. 2024.

Loshchilov, I. and Hutter, F. Decoupled Weight Decay Regularization, January 2019. URL http://arxiv.org/abs/1711.05101. arXiv:1711.05101 [cs].

Luo, C., Zhao, J., Chen, Z., Chen, B., and Anandkumar, A. MINI-SEQUENCE TRANSFORMER: optimizing intermediate memory for long sequences training. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, volume 37 of *NIPS '24*, pp. 97299–97327, Red Hook, NY, USA, June 2025. Curran Associates Inc. ISBN 979-8-3313-1438-5.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 1–15, New York, NY, USA, October 2019. Association for Computing Machinery. ISBN 978-1-4503-6873-5.

doi: 10.1145/3341301.3359646. URL https://dl.acm.org/doi/10.1145/3341301.3359646.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., and Zaharia, M. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384421. doi: 10.1145/3458817.3476209. URL https://doi.org/10.1145/3458817.3476209.

Peng, X., Shi, X., Dai, H., Jin, H., Ma, W., Xiong, Q., Yang, F., and Qian, X. Capuchin: Tensor-based GPU Memory Management for Deep Learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, pp. 891–905, New York, NY, USA, March 2020. Association for Computing Machinery. ISBN 978-1-4503-7102-5. doi: 10.1145/3373376.3378505. URL https://dl.acm.org/doi/10.1145/3373376.3378505.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. ZeRO: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20, pp. 1–16, Atlanta, Georgia, November 2020. IEEE Press. ISBN 978-1-7281-9998-6.

Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., and He, Y. ZeRO-infinity: breaking the GPU memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, pp. 1–14, New York, NY, USA, November 2021. Association for Computing Machinery. ISBN 978-1-4503-8442-1. doi: 10.1145/3458817.3476205. URL https://dl.acm.org/doi/10.1145/3458817.3476205.

Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., Li, D., and He, Y. ZeRO-Offload: Democratizing Billion-Scale model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 551–564. USENIX Association, July 2021. ISBN 978-1-939133-23-6. URL https://www.usenix.org/conference/atc21/presentation/ren-jie.

Shah, J., Bikshandi, G., Zhang, Y., Thakkar, V., Ramani, P., and Dao, T. FlashAttention-3: fast and accurate attention with asynchrony and low-precision. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*, volume 37 of *NIPS '24*, pp. 68658–68685, Red Hook, NY, USA, June 2025. Curran Associates Inc. ISBN 979-8-3313-1438-5.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer, January 2017. URL http://arxiv.org/abs/1701.06538. arXiv:1701.06538 [cs].

Tan, S., Shen, Y., Panda, R., and Courville, A. Scattered Mixture-of-Experts Implementation, October 2024. URL http://arxiv.org/abs/2403.08245. arXiv:2403.08245 [cs].

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, , and Polosukhin, I. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pp. 6000–6010, Red Hook, NY, USA, December 2017. Curran Associates Inc. ISBN 978-1-5108-6096-4.

Yang, A. e. a. Qwen3 Technical Report, May 2025. URL http://arxiv.org/abs/2505.09388. arXiv:2505.09388 [cs].

Zhao, Y., Gu, A., Varma, R., Luo, L., Huang, C.-C., Xu, M., Wright, L., Shojanazeri, H., Ott, M., Shleifer, S., Desmaison, A., Balioglu, C., Damania, P., Nguyen, B., Chauhan, G., Hao, Y., Mathews, A., and Li, S. PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. *Proc. VLDB Endow.*, 16(12):3848–3860, August 2023. ISSN 2150-8097. doi: 10.14778/3611540.3611569. URL https://dl.acm.org/doi/10.14778/3611540.3611569.

A FORWARD & BACKWARD PSEUDOCODE

Algorithm 1 AdaWS: Forward

Input: Ring capacities: $R_{\text{param}}, R_{\text{grad}}, R_{\text{act}}$, # layers: L, # Seq Groups: M, Chunks per group: $|S_0| \dots |S_{M-1}|$

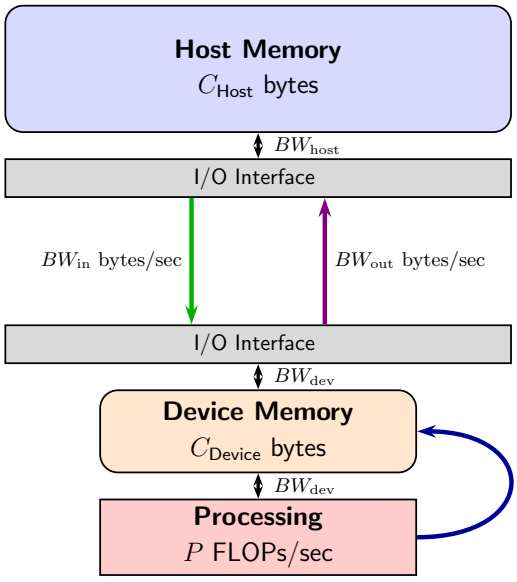
nextGrad $= L - 1$
for each layer $k = 0 \dots L-1$ **do**
  W_k = *consume*(strm$_{\text{comp}}$, parameterRing)
  for each seq group $S_i;\ i = 0 \dots M-1$ **do**
    // overwrite fwd context, F$_{\text{ctx}}$
    for each chunk $C_{j,\ S_i};\ j = 0 \dots |S_i| - 1$ **do**
      let c = global chunk index for $C_{j,\ S_i}$
      $A_{k,c}$ = *consume*(strm$_{\text{comp}}$, activationRing)
      // PERFORM COMPUTATION: use $A_{k,c}$ as canvas to write activations, update F_{ctx} in place at position j, and repopulate transition table
      T[c] = *fwd*(strm$_{\text{comp}}$, W_k, T[c], F_{ctx}, $A_{k,c}$)
      if to send $A_{k,c}$ home **then**
        // set dependency to make outbound stream wait upon completion of forward computation
        setDepend(strm$_{\text{out}}$, strm$_{\text{comp}}$)
        outbound(strm$_{\text{out}}$, H[k, c], $A_{k,c}$, save size)
        // after finishing transfer back home make this slot available for reuse
        produce(strm$_{\text{out}}$, $A_{k,c}$, activationRing)
      else
        // otherwise we can immediately repopulate for use in backwards pass
        produce(strm$_{\text{comp}}$, $A_{k,c}$, activationRing)
      end if
    end for
  end for
  // check if we need to prefetch another layer
  if $k + R_{\text{param}} < L$ **then**
    setDepend(strm$_{\text{in}}$, strm$_{\text{comp}}$)
    // overwrite current layer
    inbound(strm$_{\text{in}}$, W_k, H[$W_{k+R_{\text{param}}}$], layer size)
    // add updated layer that can be later consumed
    produce(strm$_{\text{in}}$, W_k, parameterRing)
  else
    // we will use this layer in bwd pass so readd to ring
    produce(strm$_{\text{comp}}$, W_k, parameterRing)
    // submit prefetch requests to fill gradient ring for use in bwd
    while nextGrad $\geq L - R_{grad}$ **do**
      dW = *consume*(strm$_{\text{in}}$, gradRing)
      inbound(strm$_{\text{in}}$, dW, H[dW_{nextGrad}], grad size)
      produce(strm$_{\text{in}}$, dW, gradRing)
      nextGrad -= 1
    end while
  end if
end for

Algorithm 2 AdaWS: Backward

Input: data x_i, size m
for each layer $k = L-1 \dots 0$ **do**
  W_k = *consume*(strm$_{\text{comp}}$, parameterRing)
  dW_k = *consume*(strm$_{\text{comp}}$, gradRing)
  for each seq group $S_i;\ i = M-1 \dots 0$ **do**
    // ensure F_{ctx} is populated correctly for this group
    setDepend(strm$_{\text{comp}}$, strm$_{\text{ctx}}$)
    // zero-out context gradients
    reset(B_{ctx})
    for each chunk $C_{j,\ S_i};\ j = |S_i| - 1 \dots 0$ **do**
      let c = global chunk index for $C_{j,\ S_i}$
      // retrieve forward activations
      $A_{k,c}$ = *consume*(strm$_{\text{comp}}$, activationRing)
      // PERFORM COMPUTATION: accumulate context gradients in B_{ctx}, accumulate parameter gradients in dW_k, repopulate transition table with activation gradients
      T[c] = *bwd*(strm$_{\text{comp}}$, W_k, T[c], F_{ctx}, $A_{k,c}$, dW_k, B_{ctx})
      // replace contents of F_{ctx} at position j with one needed by prior seq group or prior layer
      setDepend(strm$_{\text{ctx}}$, strm$_{\text{comp}}$)
      // either copies ctx from activation ring or retrieves from home
      updateFwdCtx(strm$_{\text{ctx}}$, F_{ctx}, k, i, j, activationRing, H)
      if to retrieve another activation **then**
        setDepend(strm$_{\text{in}}$, strm$_{\text{comp}}$)
        // overwrite current forward activations with data we will need in the future
        inbound(strm$_{\text{in}}$, $A_{k,c}$, H[next act], save size)
        // after finishing transfer mark this activation ready for use
        produce(strm$_{\text{in}}$, $A_{k,c}$, activationRing)
      end if
    end for
  end for
  // send updated gradient home
  setDepend(strm$_{\text{out}}$, strm$_{\text{comp}}$)
  outbound(strm$_{\text{out}}$, dW_k, H[dW_k], grad size)
  // check if we need to prefetch another gradient and replace current one after it is sent home
  if $k - R_{\text{grad}} \geq 0$ **then**
    setDepend(strm$_{\text{in}}$, strm$_{\text{out}}$)
    inbound(strm$_{\text{in}}$, dW_k, H[$dW_{k-R_{\text{grad}}}$], grad size)
    produce(strm$_{\text{in}}$, dW_k, gradRing)
  end if
  // check if we need to prefetch another layer
  if $k - R_{\text{param}} \geq 0$ **then**
    setDepend(strm$_{\text{in}}$, strm$_{\text{comp}}$)
    // overwrite current layer
    inbound(strm$_{\text{in}}$, W_k, H[$W_{k-R_{\text{param}}}$], layer size)
    // add updated layer that can be later consumed
    produce(strm$_{\text{in}}$, W_k, parameterRing)
  end if
end for

B AUTCONFIGURATION SCOPE

To meet our objective of flexibility across hardware environments, Transformer scales, and sequence lengths we design a general procedure that can be automatically configured based upon inputs in Figure 10.



(a) Hardware Environment

Symbol	Description
S	Maximum Sequence Length
τ	Datatype specifications
V	Vocabulary size
L	Number of layers
d_{model}	Model hidden dimension
h_{dim}	Attention head dimension
n_Q	Number of query heads
n_{ctx}	Number of key-value heads
d_{expert}	Expert network dimension
E_{shared}	Number of shared experts
E_{routed}	Number of routed experts
k	Top-k experts per token

(b) Transformer Knobs.
Note for dense models: $E_{\text{shared}} = 1$, $E_{\text{routed}} = 0$, $k = 0$

Figure 10. Inputs to AdaWS

We maintain six "structures" that track the contents of device memory: three ring buffers and three tables. The contents of the ring buffers are: (1) layer weights, (2) activation slots (for writing during forward and reading during backwards), and (3) layer gradients. The tables are for: (4) transitions between layers (i.e. residual stream, for both forward and backward) and (5 + 6) layer context windows (forward + backward).

In host memory we maintain the full training state (all weights, gradients, and optimizer states) along with space dedicated for activations generated during forward pass that will later be retrieved during backprop.

We assume user inputs memory capacities C_{dev} & C_{host}, sequence length S, and Transformer spec (10b). The system discovers processor speed and bandwidths.

Abiding by hard memory constraints is important for ideal resource provisioning and job scheduling; we haven't found similar systems that can make this guarantee.

Symbol	Description
N_{rounds}	Number of Rounds Per Step
N_{chunks}	Number of Chunks Per Round
m_{chunk}	Chunk Size (in tokens)
R_{param}	Parameter Ring Buffer Capacity $:= [1, L)$
R_{act}	Activation Ring Capacity $:= [1, L \cdot N_{\text{chunks}})$
R_{grad}	Gradient Ring Capacity $:= [1, L)$
A_{save}	$\{L\} \times \{N_{\text{chunks}}\} \to$ saved activation level

Table 3. AdaWS Dataflow Configuration