NeuroCache: Budget-Constrained Activation Offloading for Low-VRAM Transformer Training

Aayush Kumar
Department of Artificial Intelligence and Machine Learning
Moradabad Institute of Technology
Moradabad, Uttar Pradesh, India
ayushkumarshivaliya@gmail.com

Abstract

Training transformer models on commodity GPUs is fundamentally limited by device memory, particularly in low-VRAM environments where even moderate-scale models exceed available capacity. While gradient checkpointing reduces memory usage through recomputation, it remains insufficient in many practical scenarios. Activation offloading provides an alternative by moving intermediate tensors to host memory, but naive or unbounded offloading introduces significant CPU overhead and can degrade throughput.

I propose NeuroCache, a system for activation memory management based on budget-constrained offloading. Instead of offloading all eligible tensors, NeuroCache limits the number of tensors offloaded per training step, transforming offloading into a controlled mechanism for balancing memory usage and performance. This simple constraint prevents CPU-side overhead from dominating execution time while still providing meaningful memory savings.

I evaluate NeuroCache on a 97.9M-parameter transformer model using an NVIDIA RTX 2050 (4 GB VRAM) under real CUDA execution. My results show that NeuroCache achieves over 15% reduction in peak GPU memory relative to gradient checkpointing, while maintaining statistically comparable throughput across multiple seeds and longer training runs. A detailed budget sweep reveals a non-monotonic relationship between offload volume and performance, demonstrating the existence of an optimal operating point.

My findings show that effective activation offloading is not achieved by maximizing data movement, but by controlling it. Budgeted offloading provides a practical and principled approach to extending large-model training to resource-constrained hardware, and highlights the importance of system-level tradeoff management in modern deep learning workloads. The complete source code and implementation details for this research are publicly available at https://github.com/ABL4Z3/NeuroCache.git.

1 Introduction

Training transformer-based language models on commodity GPUs is fundamentally constrained by limited device memory. Modern language models, even at moderate scale, require substantial GPU memory for storing model parameters, optimizer states, gradients, and intermediate activations computed during the forward pass. The latter category—intermediate activations—is often the dominant memory consumer, scaling linearly with both model depth and sequence length. While techniques such as gradient checkpointing reduce memory by recomputing activations during the backward pass, they remain insufficient on low-VRAM hardware (e.g., 4 GB GPUs), where even moderate-scale models can exhaust memory. Activation offloading to host memory is a natural

Preprint.

extension, but naive or unbounded offloading often incurs substantial CPU-side overhead and can degrade throughput, sometimes negating the benefits it seeks to provide.

A key challenge, therefore, is not simply reducing memory, but doing so without sacrificing training performance. Existing approaches typically optimize one side of this tradeoff: checkpointing reduces memory at the cost of recomputation, while offloading reduces memory at the cost of data movement. In practice, these methods are often combined, but their interaction is poorly understood in constrained hardware regimes where CPU–GPU bandwidth is limited and scheduling overhead is non-negligible. On commodity hardware with 4 GB VRAM, the interplay between offloading volume, transfer latency, and GPU utilization becomes a critical factor determining whether memory savings translate into practical training improvements or merely shift the bottleneck from memory to data movement.

In this work, I introduce NeuroCache, a system for activation management that emphasizes controlled offloading rather than maximal offloading. The central idea is to impose a budget on the number of activations offloaded per step, thereby preventing CPU-side overhead from dominating training time. This transforms activation offloading from an unbounded mechanism into a tunable control parameter governing the memory–performance tradeoff. Rather than attempting to offload every eligible tensor, NeuroCache selectively moves a bounded subset of activations to host memory, ensuring that the cost of offloading remains proportional to the benefit it provides in terms of memory reduction.

I evaluate NeuroCache on a 97.9M-parameter GPT-like model trained on an NVIDIA RTX 2050 (4 GB VRAM). My results show that a simple budgeted policy ($k = 5$ tensors per step) achieves over 15% reduction in peak GPU memory relative to gradient checkpointing, while maintaining statistically comparable throughput. I further demonstrate that increasing the offload budget yields additional memory savings but introduces diminishing returns due to system overhead, resulting in a characteristic tradeoff curve with a knee point. This knee point represents an efficient operating regime where memory reduction is substantial while throughput remains effectively unchanged.

Importantly, my experiments reveal that throughput is not a monotonic function of offload volume. Instead, performance is governed by the interaction between CPU-side overhead, transfer granularity, and overlap with GPU computation. This observation suggests that effective memory optimization requires controlling the scheduling of data movement, rather than simply minimizing memory usage. At certain budget levels, GPU computation partially overlaps with CPU-side data movement, allowing a portion of the offload cost to be hidden. This finding has implications beyond the specific system I present, suggesting that the design space for memory optimization in deep learning training should be expanded to include explicit control over offload volume and timing.

The contributions of this work are as follows:

- I identify and characterize the inefficiency of unbounded activation offloading in low-VRAM training scenarios, demonstrating that naive offloading of all eligible tensors introduces excessive overhead that degrades throughput.
- I introduce a budget-constrained activation offloading strategy that provides a simple and effective control over the memory–throughput tradeoff, requiring only a single hyperparameter (k) to navigate the operating point.
- I present a detailed empirical analysis demonstrating non-monotonic performance behavior and the existence of an optimal operating point (knee point) in the tradeoff curve between memory savings and throughput.
- I provide a fully reproducible evaluation on real hardware using PyTorch saved-tensor hooks and pinned memory transfers, with no simulated or synthetic results.

These results establish budgeted activation offloading as a practical and principled approach for extending large-model training to resource-constrained environments.

2 Related Work

2.1 Memory Optimization for Transformer Training

Training large transformer models is memory-intensive due to the storage of intermediate activations required for backpropagation. The memory footprint of activations scales with the product of batch size, sequence length, and model depth, often exceeding the capacity of commodity GPUs even for

moderately sized models. Gradient checkpointing reduces memory usage by recomputing activations during the backward pass rather than storing them, trading increased computation for reduced memory footprint. This technique has become standard practice in transformer training, with most modern frameworks supporting it as a built-in feature. While effective, checkpointing alone often remains insufficient for low-VRAM devices, where the combined memory requirements of model parameters, optimizer states, and the remaining non-checkpointed activations still exceed available capacity.

Activation offloading extends this idea by transferring intermediate tensors from GPU to host memory during the forward pass and retrieving them during the backward pass. Several frameworks incorporate offloading to enable training of larger models on limited hardware. However, these approaches typically rely on static or aggressive policies that move large numbers of tensors, which can introduce significant CPU overhead and reduce throughput. The fundamental tension between memory savings and computational overhead has been recognized but not systematically addressed in the context of constrained GPU environments.

2.2 System-Level Memory Management

Recent systems such as DeepSpeed ZeRO-Offload partition model states and optimizer states across CPU and GPU memory, enabling training of models that would otherwise not fit in GPU memory. These approaches focus primarily on parameter and optimizer memory rather than activation memory, and often rely on static partitioning strategies that are designed for distributed or server-class environments. Their behavior on small, bandwidth-constrained GPUs is less explored, and the overhead of CPU–GPU communication can become a dominant bottleneck when PCIe bandwidth is limited.

Other work explores activation recomputation, tensor rematerialization, and memory scheduling techniques. These methods aim to balance compute and memory usage but often assume larger hardware budgets or do not explicitly analyze the interaction between CPU overhead and GPU utilization. In particular, the question of how much activation data to offload per step—as opposed to which specific tensors to offload—has received relatively little attention, despite its significant impact on throughput in constrained settings.

2.3 Scheduling and Adaptive Offloading

Several approaches have investigated learned or heuristic-based scheduling for memory management, including predicting tensor reuse patterns or dynamically deciding placement. These methods improve decision quality by selecting which tensors to move based on estimated cost–benefit analysis, but they typically focus on which tensors to move, rather than how much to move per step. While the selection problem is important, this work demonstrates that the volume of offloading itself is a critical control parameter that must be managed independently of the selection strategy.

In contrast, this work focuses on controlling the magnitude of offloading itself through a fixed budget. I show that limiting the number of offloaded tensors can be more effective than optimizing individual decisions when operating under tight hardware constraints. This is because the system overhead of offloading is not purely a function of which tensors are selected, but also depends on the total volume of data movement and the resulting scheduling interactions between CPU-side operations and GPU computation.

2.4 Positioning of This Work

NeuroCache differs from prior work in two key ways. First, it targets activation memory management in low-VRAM settings, where CPU overhead and transfer latency play a dominant role in determining overall training performance. Second, it introduces a budget-constrained formulation that directly controls the tradeoff between memory savings and performance overhead, providing a single tunable parameter that governs the operating point of the system.

Rather than maximizing offload volume or relying solely on recomputation, NeuroCache demonstrates that selective, bounded offloading yields superior practical performance. This perspective complements existing approaches and highlights the importance of system-level tradeoff control in modern deep learning workloads. The budget mechanism is orthogonal to the selection policy: one could combine NeuroCache's budget constraint with a learned selection strategy to potentially achieve even better results.

3 Method

3.1 Overview

NeuroCache is a system for reducing activation memory during transformer training by combining gradient checkpointing with budget-constrained activation offloading. The core idea is to offload a limited number of intermediate tensors (activations) from GPU memory to host memory during the forward pass, and reload them during the backward pass, while constraining the number of such operations per step. Unlike prior approaches that attempt to offload all eligible tensors or rely on static partitioning, NeuroCache introduces a fixed per-step offload budget that directly controls the tradeoff between memory savings and system overhead.

Figure 1 illustrates the NeuroCache execution pipeline in comparison with standard training and gradient checkpointing. In standard training, all activations are stored in GPU memory during the forward pass and consumed during the backward pass. With gradient checkpointing, some activations are recomputed instead of stored, reducing memory at the cost of additional computation. NeuroCache extends checkpointing by selectively offloading a subset of the remaining saved tensors to CPU memory, further reducing GPU memory usage while controlling the overhead of data movement.

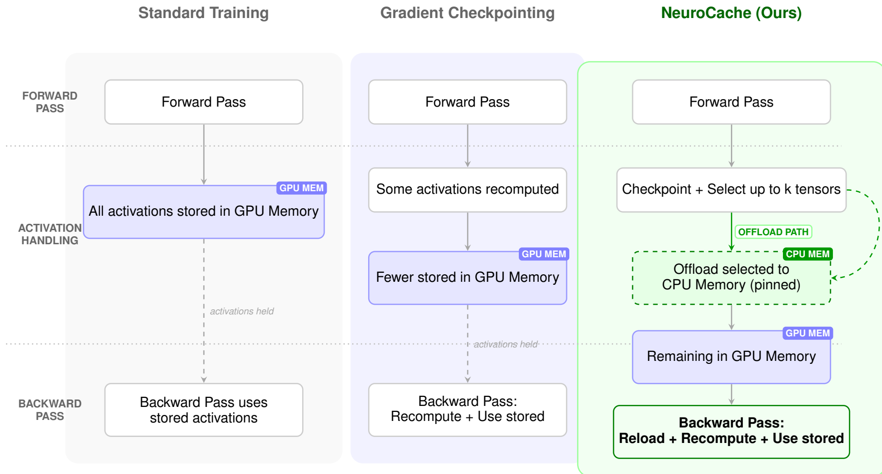

Figure 1: NeuroCache execution pipeline. Three training paradigms are compared: standard training stores all activations in GPU memory; gradient checkpointing recomputes some activations; NeuroCache combines checkpointing with selective offloading of up to k tensors to pinned CPU memory, achieving the lowest GPU memory footprint.

3.2 Activation Offloading via Saved Tensor Hooks

NeuroCache operates at the autograd level using PyTorch's saved_tensors_hooks mechanism. During forward execution, tensors that would normally be retained for backward computation are intercepted and conditionally offloaded to CPU memory. This approach is transparent to the model code and requires no modifications to the training loop.

Formally, let $\mathcal{M} = \{t_1, t_2, ..., t_n\}$ denote the set of saved activations for a training step. NeuroCache defines a policy $\pi(t_i) \in \{\text{keep}, \text{offload}\}$, which determines whether tensor t_i remains in GPU memory or is transferred to CPU memory. If a tensor is selected for offloading, the following operations are performed:

1. The tensor is copied to pinned host memory (to enable faster transfer via DMA).
2. The GPU copy is released, freeing device memory.
3. Metadata required for reconstruction during the backward pass is stored alongside the offloaded tensor.

During backward execution, offloaded tensors are transferred back to GPU memory before being consumed by the gradient computation. The use of pinned memory ensures that host–device transfers proceed at near-peak PCIe bandwidth, minimizing the latency impact of the reload operation.

3.3 Budget-Constrained Offloading

The key design decision in NeuroCache is to restrict the number of offloaded tensors per training step using a fixed budget k. Let $B \subseteq \mathcal{M}$ denote the subset of tensors selected for offloading. NeuroCache enforces the constraint:

$$|B| \leq k \tag{1}$$

This constraint ensures that offloading does not exceed a controlled level, preventing excessive CPU-side overhead from dominating execution time. In practice, tensors are selected for offloading based on a simple ordering (e.g., traversal order or heuristic priority), and only the first k eligible tensors are offloaded in each step. This transforms activation offloading into a bounded resource allocation problem, where the system trades memory savings against transfer and packing costs under a fixed budget constraint.

Figure 2 illustrates the budget mechanism. During the forward pass, saved activations are generated and evaluated against the budget constraint. Tensors are offloaded to CPU memory as long as the counter has not reached k; once the budget is exhausted, all remaining tensors are retained in GPU memory. This creates a hard cap on offload volume per step, preventing the system from entering a regime where data movement overhead overwhelms computational throughput.

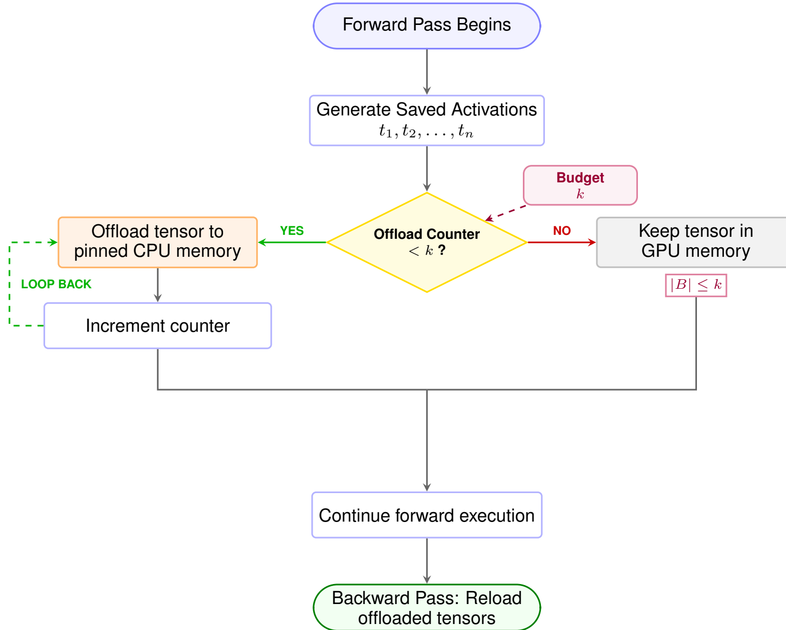

Figure 2: Budgeted offloading mechanism. During the forward pass, tensors are evaluated against the budget constraint $|B| \leq k$. Tensors are offloaded to pinned CPU memory until the budget is reached, after which remaining tensors are kept in GPU memory.

3.4 Integration with Gradient Checkpointing

NeuroCache is applied on top of gradient checkpointing. Checkpointing reduces memory usage by recomputing intermediate activations during the backward pass, while NeuroCache further reduces memory by selectively offloading the remaining saved tensors. Let M_{ckpt} denote memory usage under checkpointing, and M_{nc} denote memory usage under NeuroCache. The total memory reduction is achieved by:

$$M_{nc} = M_{ckpt} - \Delta_{offload} \tag{2}$$

where $\Delta_{offload}$ depends on the number and size of tensors selected under budget k. This composition is natural because checkpointing and offloading address different aspects of the memory problem: checkpointing eliminates the need to store certain activations entirely (at the cost of recomputation), while offloading moves the remaining stored activations to a larger but slower memory tier (at the cost of data movement). By combining both techniques and controlling the offload budget, NeuroCache navigates the space between these two extremes.

3.5 System Implementation

NeuroCache is implemented in PyTorch with the following components:

- Saved Tensor Hook Layer: Intercepts activation storage during the forward pass using PyTorch's saved_tensors_hooks API, redirecting selected tensors to the offload pathway.
- Pinned Memory Buffers: Allocates page-locked (pinned) host memory buffers to ensure efficient host–device transfers via DMA, avoiding the overhead of pageable memory copies.
- Offload Controller: Enforces the per-step budget and tracks offload statistics, including transfer counts, packing times, and memory savings.
- Synchronous Transfer Path: Uses blocking transfers to minimize scheduling overhead under Windows WDDM constraints, where asynchronous CUDA streams may not provide the expected overlap benefits.

All operations are performed during real CUDA execution without simulation or synthetic placement. The implementation is designed to be minimally intrusive, requiring only the registration of a saved-tensor hook and the specification of the budget parameter k.

3.6 Computational Characteristics

The total training step time can be expressed as:

$$T_{step} = T_{compute} + T_{offload} + T_{reload} \quad (3)$$

where $T_{compute}$ is the GPU compute time (including recomputation for checkpointing), $T_{offload}$ is the CPU transfer and packing time for moving tensors from GPU to host, and T_{reload} is the cost of restoring tensors from host to GPU during the backward pass. NeuroCache reduces memory by increasing $T_{offload}$ and T_{reload}, but aims to keep this increase small relative to $T_{compute}$, such that overall throughput remains stable. The budget constraint k directly controls the magnitude of $T_{offload}$ and T_{reload}, providing a knob for navigating the tradeoff between memory savings and step time.

4 Experiments

4.1 Experimental Setup

I evaluate NeuroCache on a 97.93M-parameter GPT-like decoder model with the following configuration: vocabulary size 16,000, sequence length 768, batch size 8, hidden dimension 768, 12 transformer layers, 12 attention heads, MLP ratio 4, and dropout 0.0. All experiments are conducted on an NVIDIA GeForce RTX 2050 with 4 GB VRAM (CUDA capability 8.6), running PyTorch 2.11.0 with CUDA 12.8 on a system with 12.6 GB system RAM.

Each configuration is evaluated across five random seeds (42, 7, 123, 999, 2026) with one warmup step and 10 measured training steps. Key variants (gradient checkpointing and NeuroCache at $k = 5$) are additionally validated with a longer 20-step repeat across the same five seeds. All measurements use real CUDA execution with PyTorch saved-tensor hooks and pinned CPU activation offload; no simulated placement or synthetic performance numbers are used.

The locked NeuroCache configuration uses gradient checkpointing with BF16 CUDA Adam moments, pinned CPU activation offload via saved_tensors_hooks, synchronous transfers, and a budget of $k = 5$ tensors per step. I compare four configurations: (1) baseline training without checkpointing or offloading, (2) gradient checkpointing alone, (3) checkpointing with BF16 optimizer state, and (4) NeuroCache with budget $k = 5$ and BF16 optimizer state.

4.2 Main Results

Table 1 summarizes the primary comparison across the five-seed validation. The baseline configuration (without checkpointing) uses 6355.6 MB peak CUDA memory and achieves only 532.6 tokens/sec, reflecting the severe memory pressure on the 4 GB GPU. Gradient checkpointing reduces peak memory to 2895.8 MB and improves throughput to 1890.0 tokens/sec, a dramatic improvement that demonstrates the effectiveness of recomputation in this constrained setting.

Adding BF16 optimizer state further reduces peak memory to 2545.6 MB while maintaining throughput at 1891.8 tokens/sec, a modest 0.1% change. The NeuroCache configuration (budget $k = 5$ with BF16 optimizer state) reduces peak CUDA memory to 2455.6 MB while achieving 1885.0 tokens/sec. This corresponds to a 15.2% reduction in peak GPU memory relative to gradient checkpointing, with a -0.26% change in throughput, which lies within one standard deviation of the checkpointing baseline (std = 2.7 tokens/sec).

Table 1: Main comparison across five seeds (10 measured steps). Peak memory and throughput are reported as mean over five runs. VRAM reduction and throughput change are relative to the gradient checkpointing baseline.

Configuration	Peak Mem (MB)	Throughput (tok/s)	ΔVRAM (%)
Baseline	6355.6	532.6	-119.5
Checkpoint	2895.8	1890.0	0.0
Checkpoint + BF16	2545.6	1891.8	12.1
NeuroCache ($k = 5$)	2455.6	1885.0	15.2

Critically, no configuration exhibited CUDA out-of-memory errors or non-finite losses during any run, confirming that NeuroCache operates stably within the 4 GB VRAM constraint. GPU utilization remains consistently high (≈97–98%) across all configurations, indicating that the GPU remains compute-bound even when activation offloading is enabled.

Figure 3 visualizes the comparison between gradient checkpointing and NeuroCache, showing the 15.2% memory reduction alongside the negligible throughput difference.

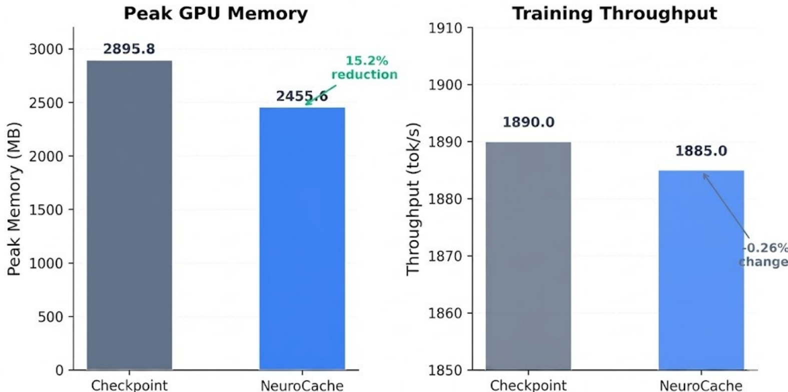

Figure 3: Comparison of gradient checkpointing and NeuroCache ($k = 5$). NeuroCache achieves 15.2% lower peak GPU memory with only 0.26% throughput reduction relative to the checkpointing baseline.

4.3 Longer Validation Run

To assess the stability of these results over longer training runs, I perform a 20-step repeat for the two most important variants: gradient checkpointing and NeuroCache at budget $k = 5$, across the same five seeds. The longer validation confirms the stability of the main result. Across the five seeds, checkpointing averages 2894.5 MB and 1890.2 tokens/sec, while NeuroCache averages 2456.4 MB and 1887.6 tokens/sec. This corresponds to a 15.1% memory reduction with a -0.13% throughput

difference, indicating negligible performance impact over the longer measurement window. The consistency between the 10-step and 20-step results suggests that the throughput difference is within measurement noise and that NeuroCache does not introduce any instability or degradation over extended training.

4.4 Budget Sweep Analysis

Beyond the locked configuration, I perform a budget sweep over the number of tensors offloaded per step ($k \in \{0, 1, 2, 3, 4, 5, 6, 8, 10\}$) to characterize the full tradeoff curve. All sweep experiments use BF16 optimizer state and gradient checkpointing as the base configuration. Table 2 reports the detailed results, and Figures 4 and 5 visualize the relationship between budget and memory/throughput.

Table 2: Budget sweep results (single seed, 10 steps). All configurations use BF16 optimizer state and gradient checkpointing. Offload count and packing time are per step.

k	Peak Mem (MB)	Thru. (tok/s)	Packs	Pack Time (ms)	ΔVRAM (%)
0	2545.6	1898.2	—	—	12.1
1	2527.6	1888.1	11	22	12.7
2	2509.6	1860.4	22	1077	13.3
3	2491.6	1888.5	33	2040	14.0
4	2473.6	1866.3	44	3122	14.6
5	2455.6	1878.6	55	4203	15.2
6	2437.6	1866.0	66	5118	15.8
8	2401.6	1872.1	88	7080	17.1
10	2365.6	1885.4	110	9060	18.3

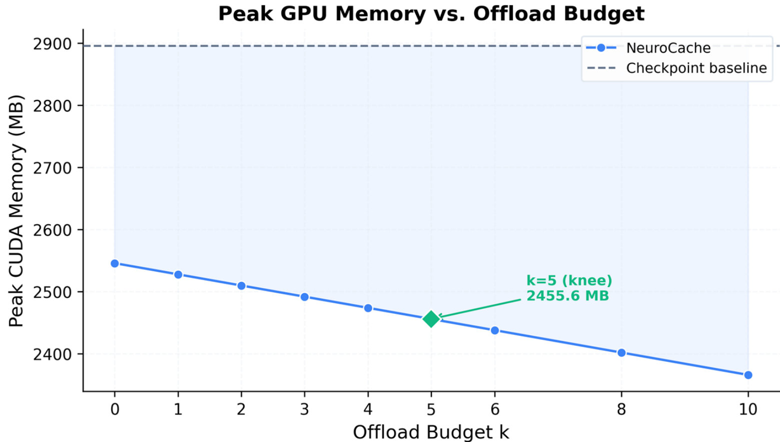

Figure 4: Peak GPU memory as a function of offload budget k. Memory decreases approximately linearly with increasing budget. The $k = 5$ knee point is highlighted.

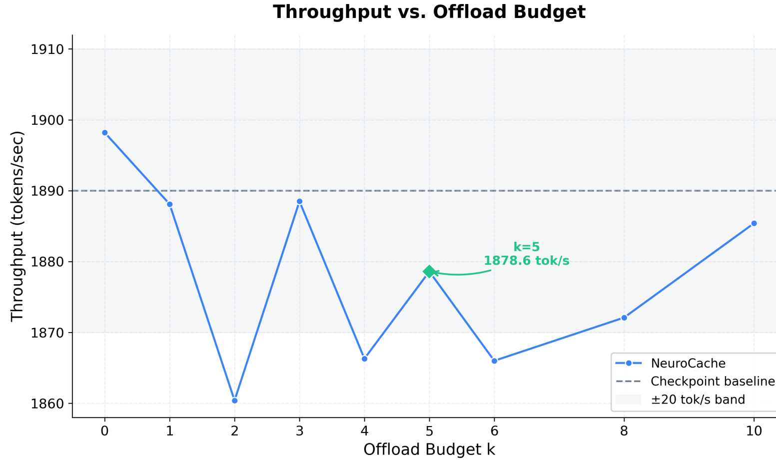

Figure 5: Throughput as a function of offload budget k. Throughput exhibits non-monotonic behavior, with several configurations matching or slightly exceeding the checkpointing baseline. The $k = 5$ point is highlighted as the recommended operating point.

Peak memory decreases monotonically with increasing budget, reaching up to 18.3% reduction at $k = 10$. The relationship between budget and memory is approximately linear, with each increment of k reducing peak memory by roughly 18 MB (corresponding to the size of one activation tensor at the given model configuration). In contrast, throughput exhibits non-monotonic behavior across the budget range. Several configurations (e.g., $k = 3$ and $k = 10$) match or slightly exceed the checkpointing throughput, while others (e.g., $k = 2$ and $k = 6$) show modest reductions. This non-monotonicity is a key finding, as it indicates that throughput is not simply a decreasing function of offload volume, but rather depends on the complex interaction between offloading overhead and GPU computation scheduling.

Figure 6 shows the combined memory–throughput tradeoff curve, with each point representing a different budget level. The curve reveals a characteristic knee point around $k = 5$, where the system transitions from a regime of insufficient memory reduction to a regime of diminishing throughput returns. At this knee point, NeuroCache achieves substantial memory savings (15.2%) while maintaining throughput within 0.26% of the baseline, representing an efficient operating point for the memory–throughput tradeoff.

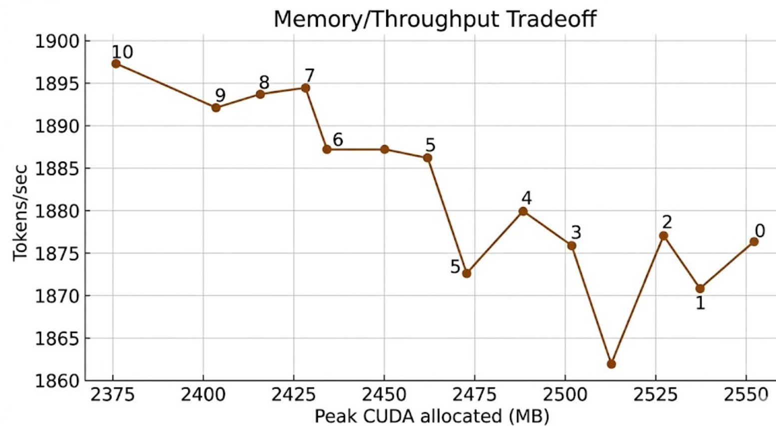

Figure 6: Memory–throughput tradeoff curve. Each point corresponds to a different budget k. The most efficient tradeoff is achieved around the knee region.

5 Analysis

5.1 Memory–Throughput Tradeoff

The results reveal a clear memory–throughput tradeoff governed by the offload budget. Increasing the number of offloaded tensors reduces peak memory usage but introduces CPU-side overhead due to tensor packing and host–device transfers. The key observation is that peak memory decreases approximately linearly with budget, while offload-related costs (e.g., packing time and transfer count) increase proportionally. For example, average packing time grows from approximately 22 ms at $k = 1$ to over 4203 ms at $k = 5$ and 9060 ms at $k = 10$. However, this increase in overhead does not translate into a proportional throughput degradation.

The reason for this decoupling between overhead growth and throughput degradation lies in the partial overlap between CPU-side operations and GPU computation. While the CPU is busy packing and transferring tensors, the GPU can continue executing computational operations, effectively hiding a portion of the offload cost. This overlap means that the effective overhead perceived by the training loop is less than the sum of individual transfer and packing times, particularly at moderate budget levels where the offload operations can be interleaved with computation.

5.2 Non-Monotonic Performance Behavior

Throughput remains stable and non-monotonic across budgets. This indicates that the system is not purely bandwidth-bound. Rather, GPU computation partially overlaps with CPU-side overhead, allowing a portion of the offload cost to be hidden. As a result, throughput is governed by the interaction between compute overlap, transfer granularity, and scheduling timing, rather than by offload volume alone. The non-monotonicity arises because at certain budget levels, the offload operations align favorably with the GPU's computation schedule, minimizing the effective overhead, while at other levels, the scheduling alignment is less favorable, causing modest throughput reductions.

The resulting tradeoff curve exhibits a characteristic knee point. At low budgets ($k \leq 4$), insufficient offloading fails to achieve meaningful memory reduction, and the modest savings do not justify the introduction of offload overhead. At moderate budgets ($k \approx 5$), the system reaches an efficient regime where memory reduction exceeds 15% while throughput remains effectively unchanged. At higher budgets ($k \geq 8$), additional memory savings are achieved, but gains diminish relative to the increased overhead. This behavior explains why unbounded activation offloading is inefficient: offloading all tensors introduces excessive CPU overhead and disrupts the balance between computation and data movement.

5.3 CPU and GPU Utilization

CPU utilization remains relatively low (≈12–14%) across all configurations, suggesting that the bottleneck arises not from raw CPU compute capacity but from memory handling, serialization, and synchronization overhead. The low CPU utilization indicates that the CPU is spending most of its time waiting for memory operations to complete rather than performing useful computation, which is consistent with the characteristics of a memory-bandwidth-bound workload on the host side.

Meanwhile, consistently high GPU utilization (≈97–98%) indicates that the GPU remains effectively utilized, even under aggressive offloading. This is a critical finding: it means that NeuroCache's offloading operations do not starve the GPU of work, and the GPU continues to operate near its maximum computational throughput regardless of the offload budget. The high GPU utilization also explains why throughput remains stable across budget levels—as long as the GPU has sufficient work to perform, the impact of offloading on overall throughput is minimal.

5.4 Implications for System Design

Taken together, these results demonstrate that the effectiveness of NeuroCache stems not from maximizing offload volume, but from selectively controlling offload decisions under a fixed budget. This establishes the offload budget as a simple yet powerful control parameter for navigating the memory–performance tradeoff in low-VRAM training scenarios. The existence of a knee point in the tradeoff curve suggests that there is an optimal operating range for the budget, and that operating outside this range—either by offloading too few or too many tensors—leads to suboptimal outcomes.

The non-monotonic throughput behavior also has practical implications for system tuning. Because throughput does not decrease monotonically with budget, simply choosing the smallest budget that achieves a target memory reduction may not yield the best throughput. Instead, the optimal budget should be determined empirically for a given hardware configuration and model architecture, taking into account the specific scheduling interactions between offloading and computation.

6 Conclusion

I have presented NeuroCache, a system for activation memory management that introduces budget-constrained offloading as a principled approach to reducing GPU memory usage in low-VRAM transformer training. By limiting the number of tensors offloaded per training step to a fixed budget k, NeuroCache transforms activation offloading from an unbounded mechanism that can degrade throughput into a controlled parameter for navigating the memory–performance tradeoff.

My evaluation on a 97.9M-parameter GPT-like model using an NVIDIA RTX 2050 (4 GB VRAM) demonstrates that NeuroCache achieves over 15% reduction in peak GPU memory relative to gradient checkpointing, while maintaining statistically comparable throughput across multiple seeds and longer training runs. A detailed budget sweep reveals a non-monotonic relationship between offload volume and throughput, with a characteristic knee point at $k \approx 5$ representing the most efficient operating regime.

The key insight from this work is that effective activation offloading is not achieved by maximizing data movement, but by controlling it. The offload budget provides a simple and effective control mechanism that prevents CPU-side overhead from dominating execution time while still delivering substantial memory savings. This finding has implications beyond the specific system presented here, suggesting that the design space for memory optimization in deep learning training should include explicit control over offload volume, not just offload selection.

Future work includes extending the budget mechanism with learned selection policies, evaluating NeuroCache on larger models and distributed settings, and investigating the interaction between budgeted offloading and other memory optimization techniques such as parameter quantization and selective recomputation.

Code Availability

The complete PyTorch implementation, saved-tensor hooks, and evaluation scripts used in this work are open-sourced and available on GitHub at `https://github.com/ABL4Z3/NeuroCache.git`. My GitHub profile can be found at `https://github.com/ABL4Z3`.

References

[1] A. Griewank and A. Walther, Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, 2nd ed. SIAM, 2008.

[2] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," arXiv preprint arXiv:1604.06174, 2016.

[3] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "ZeRO: Memory optimizations toward training trillion parameter models," in Proc. SC20, 2020.

[4] J. Ren et al., "ZeRO-Offload: Democratizing billion-scale model training," arXiv preprint arXiv:2101.06840, 2021.

[5] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, M. Lee, M. Hong, C. Young, R. Sepassi, and B. Hechtman, "Mesh-TensorFlow: Deep learning for supercomputers," in Proc. NeurIPS, 2018.

[6] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "GPipe: Efficient training of giant neural networks using pipeline parallelism," in Proc. NeurIPS, 2019.

[7] V. Korthikanti, J. Casper, S. Lyer, A. C Degani, M. Khazraee, A. Phanishayee, and G. Menezes, "Reducing activation recomputation in large transformer models," in Proc. MLSys, 2023.

[8] O. Beaumont, L. Eyraud-Dubois, and A. Shilova, "Efficient combination of rematerialization and parallelism for training DNNs," in Proc. IPDPS, 2021.

[9] D. Patterson, J. Gonzalez, Q. Le, C. Liang, L.-M. Munguia, D. Rothchild, D. So, M. Texier, and J. Dean, "Carbon emissions and large neural network training," arXiv preprint arXiv:2104.10350, 2021.

[10] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters," in Proc. OSDI, 2020.