Buffer-Level Residency Management for GPU Memory Extension

Streaming a network through a device that cannot hold it, without telling the network

Aldric Pierrain
Cydral Technology, Paris, France
contact@cydral-technology.eu

8 September 2026

Abstract

Training and inference for modern neural networks are bounded less by arithmetic than by the capacity of a single accelerator. A decade of work has answered this by managing tensor residency inside the framework executor, where the dataflow graph makes tensor identity, liveness and read or write intent available for free. We ask what remains possible one level lower, inside the buffer object itself, where none of that information exists. We describe a residency manager that streams blocks between device memory, pinned host memory and a memory-mapped store under a budget, driven only by the sequence of accesses the run reveals about itself, and that reconstructs the missing dirty bit by fingerprinting each block on the device rather than moving it across the link to find out whether it changed. The mechanism requires no modification to any network, layer or training loop; it is a runtime option, and when it is off the accessor path costs a single predicted branch. We report measurements from an implementation inside a C++ deep learning library, where it costs 1.4 times the resident case in training and 2.7 in inference on a budget well below the working set, and describe two defects in that library which the instrumentation built for this work exposed. Buffer-level residency extends the working life of the accelerators most practitioners already own; unified-memory hardware answers the same shortage by removing it, and where that hardware is at hand it is the better answer.

Keywords: *GPU memory management, memory oversubscription, tensor offloading, prefetching, access pattern prediction, deep learning systems, frugal computing*

1. Introduction

The memory of a single accelerator has become the binding constraint on what a practitioner can train. A model whose weights fit comfortably may still fail the moment the context window is widened or the batch is enlarged, because gradients, optimizer states and activations scale with those choices while the card does not. The gap is widest exactly where it matters most for accessibility: on the eight to twelve gigabyte devices that ship in laptops and entry-level workstations, which is what most people actually have.

The established answer is to manage residency: keep in device memory only what a kernel is about to read, and hold the rest elsewhere. Ten years of systems work has refined this, and it has done so consistently at one level of the stack, inside the framework executor, where the dataflow graph is available. That placement is natural and it is productive, because the graph answers for free the three questions a residency policy needs: which tensor is this, when will it next be used, and will the next use read it or write it.

This paper asks a different question. What can a residency manager do if it sits below the graph, inside the buffer object that every tensor ultimately owns, where none of those three answers is available. The mechanism built to answer it is called V^{2}RAM, for video memory virtualised in software. The motivation is practical. A manager at that level is indifferent to the framework above it. It works for training and inference alike, for any topology, and for code the framework authors never anticipated, because it never learns what any of it means. The price is that it must recover, at runtime and from observation alone, information that a graph-level manager is simply handed.

We make three contributions. First, we show that the first two questions, identity and next use, can be answered by observing the access sequence, and that a network's regularity makes this reliable enough to drive both prefetching and an optimal replacement policy, while its absence is detectable and degrades safely. Second, we show that the third question, read or write, admits an answer that graph-level managers never needed and that is cheap for a reason specific to accelerators: reading a block where it already sits costs a fraction of moving it to find out what it holds. Third, we report honest measurements, including the cost of the mechanism against the case where it is not needed, and we describe what the work revealed about the host library itself.

2. Prior work

Residency management for deep learning has a well-defined lineage, and this work belongs inside it.

vDNN [1] established the shape of the solution. It offloads inter-layer activation maps to host memory during the forward pass and prefetches them back before the backward pass reuses them, through a dedicated stream that allocates pinned host memory and issues non-blocking transfers over PCIe. Reported savings reach ninety percent of average device memory on convolutional networks, at roughly eighteen percent throughput loss against a hypothetical device large enough to avoid offloading altogether. One detail of its design is central to this paper: vDNN can overlap its transfers safely because activation

maps are read-only during the forward pass, and it knows this from the graph.

SuperNeurons [4] and moDNN [7] refined the policy, the latter triggering offload when an allocation fails and recording that an already-offloaded block need not be written again because the host retains its copy. SwapAdvisor [3] planned swapping by searching over the dataflow graph, and later work automated the same decisions inside a production framework [9]. These systems share vDNN's position in the stack and its reliance on static analysis.

Capuchin [2] is the closest antecedent to the present work and deserves precise statement. It manages memory per tensor, not per layer, combining eviction, prefetching and recomputation, and its defining feature is that decisions come from the tensor access pattern tracked at runtime. The design is motivated by the observation that this pattern is regular across training iterations, and execution is split into a measured phase, the first minibatch, from which the access sequence is taken, and a guided phase driven by the policy that sequence produced. Our observation thread, our schedule and our prefetch horizon are the same idea. Capuchin, however, is a module inside a framework and identifies tensors by a unique identifier the framework provides; it operates above the level we are concerned with.

ZeRO-Offload [6] and ZeRO-Infinity [5] address the same shortage from the distributed training side, partitioning parameters, gradients and optimizer states across device, host and NVMe storage. ZeRO-Infinity's three-tier hierarchy corresponds to ours, but it is driven by an explicit input-output engine and by partitioning decisions taken with full knowledge of the model.

Two further points bound the novelty of anything in this area. A granted United States patent [7] covers methods of operating a graphics processing unit to train a deep neural network using local memory, including the offload and prefetch scheduling and the observation that a block once offloaded need not be written again. And a recent preprint on fault-tolerant inference [11] reports device-side change detection for checkpointing, comparing against a device-resident shadow copy at HBM bandwidth and transferring only the pages that differ, which is the same cost argument we make below. Its shadow copy doubles device memory, which is acceptable for checkpointing and self-defeating for a system whose purpose is to reduce device memory.

What we have not found described is a residency manager placed below the graph, and the technique that placement forces: recovering the dirty bit by fingerprint rather than by knowledge or by shadow. We state this as an absence in the literature we surveyed, not as a claim of priority.

System	Level	Next use from	Intent
vDNN (2016)	graph	static analysis	known
SuperNeurons (2018)	graph	liveness	known
moDNN (2018)	graph	static schedule	assumed
SwapAdvisor (2020)	graph	graph search	known
Capuchin (2020)	tensor	runtime trace	known
ZeRO-Infinity (2021)	states	partition plan	known
This work	buffer	runtime trace	inferred

Table 1. Residency managers by the level at which they operate and the information available there.

3. What the position costs, and what it buys

A buffer-level manager cannot ask the three questions a graph-level manager takes for granted. It sees allocations, sizes and accesses, and nothing else. Figure 1 places the two designs against each other.

Identity is the easy one: a buffer has an address and a lifetime, so a stable identifier can be assigned at allocation and retired at destruction. Next use is harder but tractable, and section 4 addresses it. Read or write intent is the one that has no local answer at all, and section 5 is about what to do instead.

Against these costs, the position buys three things. The manager is indifferent to what runs above it, so a single implementation serves training, inference, fine-tuning and any topology, including ones written after it. It requires no modification anywhere in the framework's layers or in user code, which matters because the value of a memory system is bounded by how much code is willing to adopt it. And it composes with the framework's own coherence protocol instead of replacing it, which keeps the correctness argument small enough to check.

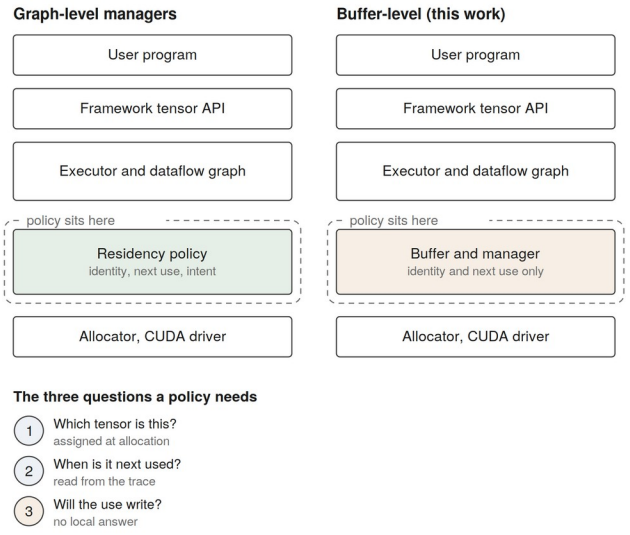


Figure 1. Where a residency policy is usually placed, and where this one is. The right-hand stack is unmodified above the buffer object.

4. Residency from an observed schedule

The manager holds each block in one of three tiers, shown in Figure 2. On the device it is readable by a kernel. In the host tier its device buffer has been released and a pinned mirror holds it.

In the store tier both buffers are gone and the content lives in a slot of a memory-mapped sparse file.

The store deserves comment because the choice of mapping rather than explicit input and output is deliberate. The pages of a mapping are page cache: while host memory is available they stay resident and the tier runs at memory speed, and when it is not the kernel writes them back and reclaims them. The same tier is therefore a host-memory cache on a machine with room and a disk overflow on a machine without, with no threshold to choose. Explicit buffers cannot behave this way, because memory the program holds is memory the kernel may not reclaim. A second consequence is that a block in the store needs no copy to be read on the host: the accessor returns a pointer into the mapping, so loading a model writes through to the store rather than into pinned buffers that must then be spilled.

What a mapping cannot do is feed a DMA engine well, its pages being pageable. Transfers between the store and the device therefore pass through a small pinned buffer in chunks, with the host copy of one chunk overlapping the transfer of the one before it. That buffer is the only pinned allocation whose size does not follow the model.

An observation thread reads a lock-free trace of accesses, searches it for a period, and publishes a schedule once a candidate has been confirmed over two full repetitions (Figure 4). Two details decide whether the search finds anything real. Candidate periods are generated from the gaps between repetitions of the rarest block in the window rather than the most recent access, because the last thing read before a step boundary is often a normalization scale that every layer also reads, whose gaps are layer spacings and never the step. And a candidate claiming more than a small number of accesses per distinct block is rejected: a step reads each tensor it touches a few times, so a longer period is a coincidence the trace happened to confirm.

A published schedule serves three purposes. It says which block is coming, so the observation thread can bring it in ahead of use. It gives the distance to each block's next use, which makes eviction by furthest next use available. That is the optimal replacement policy, known to be so since Belady established it for virtual storage in 1966 [14], and no mechanism without foresight can implement it. And it drives a deeper horizon on which the store's pages are hinted into host memory, so that a block's pages are resident before the transfer that reads them begins.

When no period verifies the schedule is simply not published, eviction falls back to least recent use and nothing is prefetched. This is not a rare corner, and our measurements separate the cases cleanly. On a network whose feed-forward runs a data-dependent number of steps, no period exists to be found and none is: the manager says so and falls back. On the same network with that layer replaced by a fixed one, a period of 1400 accesses across 285 blocks is identified within the first few steps and holds for the rest of the run, and the observation thread then moves half a gigabyte alongside the computation without charging the program for any of it. Correctness never depends on a schedule being found; only the quality of the policy does.

One property carries the concurrency of the whole design: a compute thread is the only party that lowers a block's tier, and the observation thread only raises one, and only into headroom the budget already leaves. A block on the device therefore stays there for as long as the thread that put it there is looking at it, which is what allows the common case to run without taking a lock. The single exception, releasing memory after a period of inactivity, is safe for the same reason that makes it useful: nothing has been accessed, so no kernel is in flight.

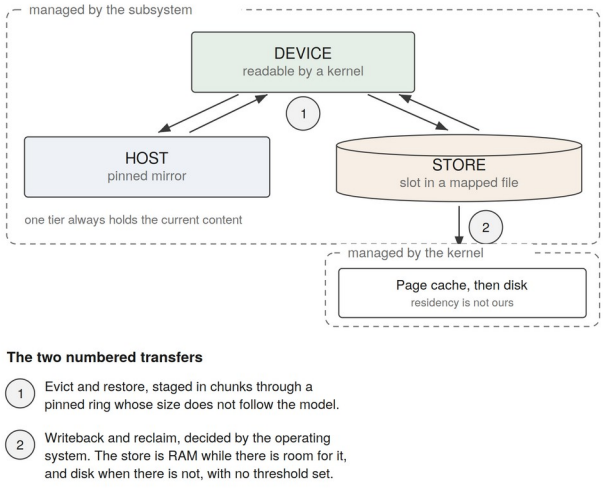


Figure 2. The three tiers and the invariants that hold between them. The store is a mapping, so its residency in host memory is the kernel's decision.

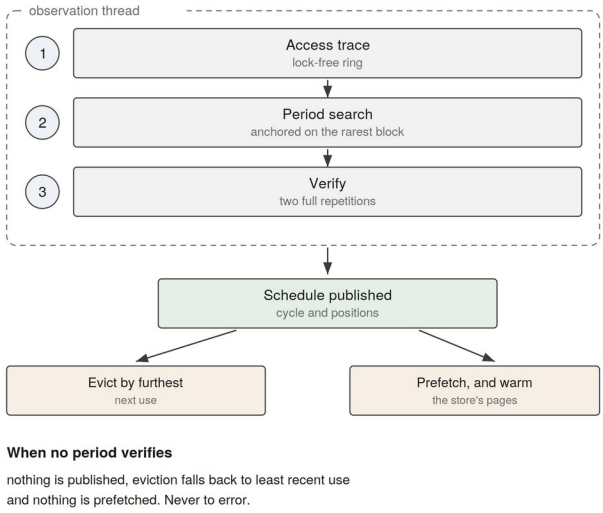


Figure 4. The observation thread. A schedule is published only after a candidate period has been confirmed over two repetitions.

5. Reconstructing the dirty bit

The accessor that hands out a device pointer cannot know whether the caller will read through it or write through it. Const-ness is a weak signal and in practice a misleading one: a framework hands a layer its parameters through a non-const

accessor because the layer owns them, not because it intends to modify them. A manager with no better information must assume every device access is a write, and therefore that every eviction requires the block to be moved back to the tier below it.

For inference this assumption is almost always wrong and almost always expensive. The weights of a model are read on every step and modified on none, yet each eviction pays a full transfer down the link to preserve bytes that did not change.

The alternative is to determine, rather than assume, and the cost argument is specific to accelerators. Reading a block where it already sits proceeds at device memory bandwidth, several hundred gigabytes per second; moving the same block to find out what it holds proceeds at link bandwidth, roughly an order of magnitude less. A fingerprint computed on the device therefore replaces a transfer with a comparison of sixteen bytes (Figure 3).

The fingerprint is a pair of sixty-four bit accumulators over the block read as thirty-two bit words. Each word is mixed with its own index through a finalizer, so a permutation is not invisible, and the mixed values are combined both by exclusive-or and by addition. Both combiners are associative and commutative, which allows the reduction to proceed in whatever order the scheduler chooses across threads and blocks; using two of them costs nothing and makes the pair behave as a hundred and twenty-eight bit value. The cross-block reduction is completed on the host from partial results, since intra-block synchronization orders threads within one block and nothing beyond it.

Two blocks with different contents therefore collide with probability of order two to the minus one hundred and twenty-eight, which is many orders of magnitude below the probability of an undetected memory error on the same data. We validated the construction on the properties the eviction path depends on: identical data yields an identical fingerprint, a one-unit-in-the-last-place change to a single word is detected, two thousand single-bit flips at random positions are all detected, a transposition of two words is detected, and no collision occurred over twelve and a half million pairs of distinct blocks. Twelve grid geometries reduced in shuffled order all agreed with a serial reference.

The threshold below which fingerprinting stops paying cannot be chosen in advance. A fingerprint costs a kernel launch, the same fixed amount whatever the block holds, while moving the block costs its size divided by the link, and the two cross at one launch worth of bandwidth, a quantity that depends on the device, the driver and where the store sits. We fixed it at four megabytes on a first attempt and doubled the traffic of a run; the manager now measures both terms and reads the threshold off them, converging to between 634 and 654 kilobytes across three independent runs on our test machine, and settling within a few percent of that figure on every configuration we measured afterwards.

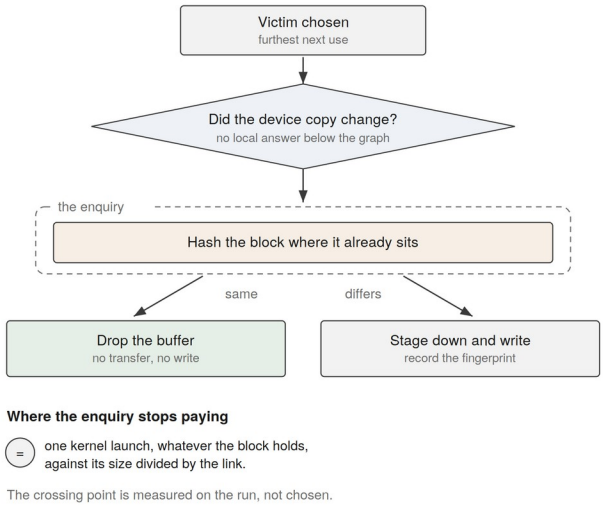


Figure 3. The eviction path. The question in the second box has no local answer below the dataflow graph.

6. Implementation and evaluation

V²RAM was implemented inside Dlib [13], a C++ machine learning library whose deep learning module routes every device allocation through a single buffer class. That library is a means here rather than an end: what it provides is a realistic setting in which to measure whether such a mechanism can be added to an existing stack without disturbing it, and how it behaves on real training and inference workloads. The implementation touches the buffer class, the allocator and a new manager; no layer, network definition or training loop was modified.

When the mechanism is not enabled, each accessor performs one test on a null pointer. Measured over four hundred thousand accessor calls, this costs 1.47 nanoseconds per call, which is the cost of a predicted branch, and the library's behaviour is unchanged.

All measurements below were taken on a laptop-class device with twelve gigabytes of memory over a PCIe link, with the store placed on a memory filesystem so that the speed of the backing volume does not enter the comparison, and each configuration run three times with the median reported. Runs where the store sat on slower storage were dominated by writeback from preceding runs and are excluded, since they measure the volume rather than the mechanism.

Table 2 gives the cost of the mechanism against keeping everything resident, on two models. The figures are the price of capacity, not a regression: in both cases the model fits the device and the budget was set below it deliberately, to force the mechanism to work.

Table 3 decomposes a step. Transfers on the critical path run at close to what the link provides, which says that the remaining cost is the bytes themselves and not the way they are moved. The volume is already at its floor: a working set of about 200 MiB under a 128 MiB budget must bring in at least 72 MiB per step, and the measured figure is 70.

Table 4 isolates the fingerprint. With it disabled, every eviction reads its block back over the link to preserve content that, for a generation loop, never changes.

Two results deserve separate mention because neither was anticipated. The first is that the mechanism costs less in training than in inference, 1.4 against 2.7 times the resident case, because a training step performs far more arithmetic per byte it moves. This is the reverse of what we expected and it favours the case the approach exists to serve.

The second is that enlarging the budget can make a training run slower. Raising it by half, while widening the window that protects recently touched blocks, cost 1.8 times the time and multiplied the traffic returning from the device by three and a half. A block that stays resident longer is more likely to have been written between two evictions, so the fingerprint stops paying: the share of evictions it settled fell from 43 percent to 3. A cache normally improves with size; a residency manager whose economy rests on recognising unchanged blocks does not, and the budget has an optimum rather than a direction.

Configuration	Budget	Resident	With manager	Ratio
Inference, 200 MiB set	128 MiB	13.2 ms	35.2 ms	2.7x
Training, 1186 MiB set	128 MiB	256.6 ms	353.7 ms	1.4x

Table 2. Milliseconds per step, median of three runs, store on a memory filesystem. The budget is set below the working set on purpose.

Component	Share of a step
Compute, from the resident run	37.5 %
Waiting on transfers, at 7.0 GiB/s	49.0 %
Fingerprinting	8.0 %
Device synchronisation	4.1 %
Locks and bookkeeping	1.4 %
Prefetch, concurrent, not charged	0 %

Table 3. Where a 35.2 ms inference step goes. Prefetch runs on the observation thread and is not charged to the step.

	Fingerprint off	Fingerprint on
Evictions settled with no transfer	0 of 3512	1282 of 3512
Moved per run	25 005 MiB	12 419 MiB
Time per step	45.7 ms	35.2 ms
Threshold	not in force	634-654 KiB

Table 4. Effect of the fingerprint on inference, same session, same budget. The threshold is read off the run rather than chosen.

7. Instrumentation as a by-product

A residency manager sees every tensor access a program makes, which makes it an instrument as well as a mechanism, and the instrumentation built to tune it turned out to answer questions about the host library that framework-level profiling had not raised. We give two examples briefly, not because they belong to the memory system, but because they illustrate a use for it that we did not anticipate.

Counting, per block, the host accesses that force a transfer back from the device exposed a dead round trip inside an element-wise CUDA operation: host pointers were retrieved and never read, a remnant of a CPU implementation of the same function, and one of them invalidated the device copy so that the following kernel pushed the same bytes back. Every call moved its destination across the link twice for nothing.

The same counter pointed at an adaptive computation layer [12], which lets a network learn how many steps to spend on each position. Its halting logic, a small per-position update, executed on the host, and its backward pass performed element-wise loops over full state tensors there. On a representative configuration this crossed the link 1.41 GiB per training step; moving the logic onto the device reduced that to 288 bytes. Auditing the layer also exposed an error unrelated to placement, in which a network applied at several steps received the parameter gradient of one step rather than their sum.

Neither finding concerns memory extension, and both affect runs with the mechanism disabled. What they illustrate is a general point: a component that must observe every access in order to place data can report, at no additional cost, where a program is spending its link, and it does so at a granularity that sits below what a framework profiler reports.

Measure	Before	After
Host accesses over 100 steps	12 792	3 224
Bytes per access	33 KiB	3.9
host() calls in the layer	23	3
Transition gradient	last step	sum of steps

Table 5. Adaptive computation layer, before and after moving its state machine onto the device. The residual accesses are a four-byte counter read once per step per layer, which is the early exit from the step loop.

8. What the mechanism buys

The measurements above say what the approach costs. What it is for is capacity, and that is measured by asking how large a working set a given card will carry. We trained a 270 million parameter grouped-query model with four experts per layer on a 12288 MiB device, raising the batch until the run no longer completed, with a budget of 8192 MiB given to the manager.

With the mechanism absent the run does not start at all, at any batch size. The library gives every tensor a device buffer and a pinned host copy at construction, so a working set of six gigabytes asks for six on the card and six more in host memory that can never be paged out; the host is what runs out first. Under the manager neither copy is taken until something needs it.

With the mechanism the working set grows linearly with the batch, 4416 MiB of weights, gradients and optimizer moments that do not depend on it, and 1637 MiB of activations per unit of batch. The largest carried was 17506 MiB: 2.14 times the budget the manager was given and 1.42 times the card it ran on. The next step up was killed, its working set exceeding the host memory rather than the device.

Two figures in that table deserve attention. The peak occupancy of the card falls as the batch rises, from 10748 MiB down to 9092, because a larger working set leaves less room for the buffer pool and the manager gives that room back before it evicts anything live. And the ceiling is not the device: at batch 16 the run died on host memory, which is the honest boundary of a mechanism whose lower tiers are the host and a file on it.

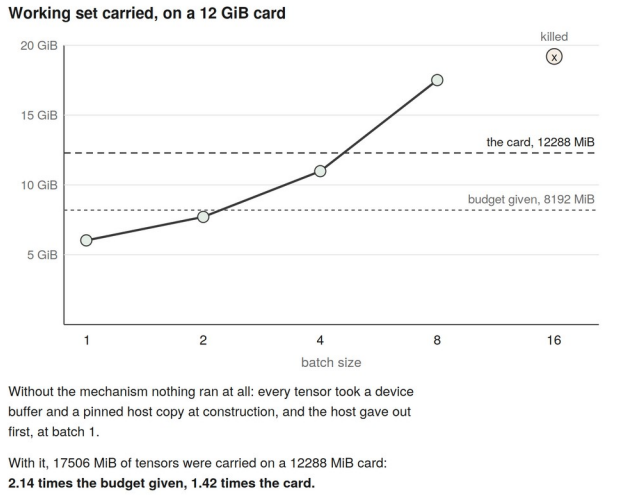


Figure 5. Working set carried against batch size. The mechanism holds a working set larger than the card by keeping resident only what a kernel is about to read.

Batch	Working set	Peak on card	Outcome
none	6028 MiB	did not start	host out of memory
1	6028 MiB	10748 MiB	ran
2	7709 MiB	10170 MiB	ran
4	10984 MiB	9568 MiB	ran
8	17506 MiB	9178 MiB	ran
16	30600 est.	9092 MiB	killed

Table 6. A 270 M parameter model trained on a 12288 MiB card with a budget of 8192 MiB. The first row is the same run with the mechanism absent.

9. Limitations

V²RAM extends capacity and does not improve throughput. On a model that fits, it costs between two and nine times the resident case depending on how far the budget is set below the working set, and that cost is close to irreducible: the transfers already run at link speed and the volume is near its theoretical floor.

The predictive half is inert on networks whose control flow depends on their input, and the model of the previous section is one: a mixture of experts routes each token to two of four, so which weights a step touches is a function of the data and no period exists to be found. A layer whose depth adapts to its input does the same. The manager detects the absence, declines to publish a schedule and falls back to least recent use, so nothing breaks; but eviction by furthest next use and prefetching are both unavailable exactly where the capacity matters most. This is a property of the principle rather than a defect of the implementation, and the responsibility runs the right way: a layer that wants to be prefetched should present a stable access sequence.

Where a schedule is found, it earns less than it might. Between five and seven percent of restores were anticipated in our inference runs, against a horizon deliberately kept short. The prefetcher competes for the same link as the transfers the program is waiting on, so fetching further ahead would take bandwidth from work already known to be needed. Where the two should be balanced is a question our measurements pose rather than answer.

The budget can only be honoured down to the total size of blocks that are never evicted. A threshold protects small tensors, whose transfer cost dominates their footprint, and lowering it to reclaim their space made a run slower by making activations evictable.

A display driver that pages removes the clean failure the mechanism relies on for its own diagnosis. An allocation past the physical memory is met in host memory rather than refused, so a budget set above what the device reports free is silently honoured somewhere else and the run merely slows. The manager warns when the budget exceeds the free figure at startup, which is all it can do from inside the runtime.

The manager governs one device. Blocks allocated elsewhere are left alone. Activation is a one-time decision taken before any tensor exists, and there is no way to switch the mechanism off during a run, because doing so would require bringing every block back at once, which is precisely the operation the mechanism exists because one cannot perform.

Two levers this work does not pull deserve to be told apart, because only one of them is closed to a manager in this position. Recomputation, which SuperNeurons [4] and Capuchin [2] weigh against moving a tensor and often prefer, requires knowing which operation produced it. That is a fact about the graph and there is no way to recover it from a sequence of accesses, so it is genuinely unavailable here.

Compression is not. A codec between the device buffer and the store slot reads bytes and writes bytes; it needs to know nothing about what they mean, so it sits entirely inside the manager and costs the transparency nothing. Earlier work applied it to activation sparsity [8] and to swapped tensors generally [10]. What our measurements do not establish is that it would pay: the ratio a lossless codec achieves on floating-point tensors is poor except where they are sparse, which is why the work above targeted activations rather than weights, and a codec must run faster than the link it spares. One observation makes the question cheap to settle. The fingerprint pass already reads every byte of a block on the device, at device bandwidth, before any transfer; it could report how compressible the block is at almost no additional cost, and the decision to compress could then be taken per block on measurement rather than by assumption.

Reducing precision at rest, storing a block in half the width and widening it on the way back, is a different proposition again. It is no more graph-dependent than a lossless codec, so it raises no question of transparency, but it changes the values a program reads and therefore the arithmetic it performs. That belongs to the caller rather than to the memory system, and would have to be asked for explicitly rather than applied because it happened to save bandwidth.

Finally, the fingerprint is probabilistic. We consider a collision probability of order two to the minus one hundred and twenty-eight acceptable against the alternative, but it is not zero, and a verification mode that checks each decision by full comparison is provided for exactly that reason.

10. Code availability

The implementation described here is part of a public fork of the Dlib library and is distributed under the Boost Software License, at github.com/Cydral/Dlib-Transformer-extensions. It comprises the residency manager, its CUDA kernels, the two backends required for the library's rule that behaviour be identical on host and device, and a runnable example that reproduces the measurements reported above through command-line options for the budget, the store location, the prefetch horizon and the fingerprint. The verification mode referred to in section 8 is exposed through the same interface, so the correctness of every fingerprint decision can be rechecked on any model by full comparison.

11. Conclusion

Unified-memory systems will dissolve most of what this paper addresses. Where the processor and the accelerator share one physical memory, residency ceases to be a question and the training of large models on a single machine becomes a matter of bandwidth rather than capacity. That hardware exists, and it works.

It is also expensive and uncommon, and the trend in component prices is not encouraging. The practical consequence is that for some years yet the majority of people who want to train or fine-tune a model will do so on a device with eight to twelve gigabytes. What can be made to run there therefore determines who participates, and it will keep doing so for as long as that is the hardware in most hands.

We have argued that a residency manager can be placed lower in the stack than convention puts it, that the information it loses there can be recovered from observation and from the accelerator itself, and that the resulting mechanism is transparent enough to be adopted without changing any model. The cost is real and this report measures it. What the approach offers is reach. It moves the boundary of what a modest machine can attempt, and it charges time for doing so.

That seems worth doing on its own terms. A field whose progress depends on hardware that fewer and fewer people can buy is a field that narrows. Frugality keeps innovation in more hands, which is why it is worth designing for.

References

[1] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. In Proc. MICRO, 2016.

[2] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, and X. Qian. Capuchin: Tensor-based GPU Memory Management for Deep Learning. In Proc. ASPLOS, pp. 891-905, 2020.

[3] C.-C. Huang, G. Jin, and J. Li. SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In Proc. ASPLOS, pp. 1341-1355, 2020.

[4] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska. SuperNeurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In Proc. PPoPP, 2018.

[5] S. Rajbhandari, O. Ruwase, J. Rasley, S. Smith, and Y. He. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. In Proc. SC, pp. 1-14, 2021. arXiv:2104.07857.

[6] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He. ZeRO-Offload: Democratizing Billion-Scale Model Training. In Proc. USENIX ATC, pp. 551-564, 2021.

[7] X. Chen, D. Z. Chen, and X. S. Hu. moDNN: Memory Optimal DNN Training on GPUs. In Proc. DATE, 2018. See also US Patent 11,599,798 B2, Methods of operating a graphics processing unit to train a deep neural network using a GPU local memory.

[8] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. In Proc. HPCA, 2018. arXiv:1705.01626.

[9] T. D. Le, H. Imai, Y. Negishi, and K. Kawachiya. Automatic GPU Memory Management for Large Neural Models in TensorFlow. In Proc. ISMM, pp. 1-13, 2019.

[10] P. Chen, S. He, X. Zhang, S. Chen, P. Hong, Y. Yin, X.-H. Sun, and G. Chen. CSWAP: A Self-Tuning Compression Framework for Accelerating Tensor Swapping in GPUs. In Proc. IEEE CLUSTER, 2021.

[11] Concordia: JIT-Compiled Persistent-Kernel Checkpointing for Fault-Tolerant LLM Inference. Preprint, arXiv:2606.23521, 2026.

[12] A. Graves. Adaptive Computation Time for Recurrent Neural Networks. Preprint, arXiv:1603.08983, 2016.

[13] D. E. King. Dlib-ml: A Machine Learning Toolkit. Journal of Machine Learning Research, 10:1755-1758, 2009.

[14] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. IBM Systems Journal, 5(2):78-101, 1966.