Training That Survives Reclaim

One snapshot on notice loses no steps, periodic snapshots lose to file checkpoints, and a compiled trainer resumes in 6.6 s against 28.1 s

Pranav Varshney
Nodus Compute

Viswa Kotra
Nodus Compute

Abstract

Reclaimable and idle GPU capacity adds AI work per GPU and per fleet only if the training it runs survives being reclaimed. Nodus, the infrastructure execution layer for AI workloads, measured whether a GPU memory snapshot of a live training process can serve as that recovery, against the standard file checkpoint. An 85.35 million parameter PyTorch GPT trained on two GPUs on two hosts through the Nodus node runtime's guarded runc executor, with CRIU and NVIDIA cuda-checkpoint taking and restoring the snapshots. A snapshot frozen mid-step restored on the other host's GPU and continued bit-identically, with equal loss bits at every step and equal SHA-256 digests of the parameters, optimizer state and random number state, in 3 restores of the eager trainer and 1 of a torch.compile trainer. The snapshot is 8.53 GB, because it holds PyTorch's whole 7.14 GB memory reserve, against a 1.02 GB checkpoint file, and each snapshot stopped training for 12.9 s against 1.36 s for a checkpoint. At a 60 s cadence that slowed training by 24.3% against 2.4%. In a reclaim drill 270 s into the job, the snapshot arm lost 278 steps against 123, and finishing the job took 28.8% more GPU-seconds than an uninterrupted run against 8.7%. Periodic snapshots lose to periodic file checkpoints. A single stopping snapshot taken on a reclaim notice lost no steps, and the first new step ran on the other host 44 s after the freeze, of which about 37 s had to fit inside the notice. Snapshots also win when restarting is expensive: the torch.compile trainer reached its first step 6.6 s after a restore against 28.1 s for a file resume that compiles again. A cost model computed from these measurements puts periodic snapshots at 9.6% overhead at one reclaim per hour, periodic file checkpoints at 3.0% and one snapshot on notice at 1.2%. A compile worker that held an NVML handle blocked the first snapshot of the compiled trainer. The results come from one drill per arm, one model size and host-to-host copies, and no independent verifier has rerun them. A follow-up that released PyTorch's free reserve at a step boundary before each snapshot cut the snapshot to 3.69 GB and the restore to 3.74 s and the notice needed to 23.8 to 29.2 s with no lost steps, but periodic snapshots still lost to file checkpoints, 9.1 s against 1.44 s of stall per protection point. A second follow-up, on one GPU under a compressed reclaim schedule, found that the snapshot on notice, now in the product for GPU Jobs behind a per-org Beta flag, raised goodput from 0.724 to 0.848 of an uninterrupted run, that setting the file-checkpoint cadence from unannounced failures alone added a smaller and noisier 1.2 points, to 0.860, and that without file checkpoints an unannounced kill cost up to 2384 steps.

1 Introduction

More AI per GPU and per fleet depends on putting idle and reclaimable capacity to work. A GPU that serves inference at peak and waits off-peak, or one held for reserved work that has not arrived, can train in the meantime, but only if the training survives the moment the GPU is taken back. A job that loses its progress at every reclaim spends the borrowed capacity on work it then repeats. Training jobs protect themselves with file checkpoints, which save the model and optimizer state every few minutes and resume from the newest one. Our reports 033 [1] and 034 [2] showed that a containerized inference engine comes back from a GPU memory snapshot in seconds. This report asks whether the same kind of snapshot works as reclaim recovery for a training process, and whether it does better than the file checkpoint.

We trained one PyTorch [3] model on two GPUs on two hosts and protected it in three ways: a GPU memory snapshot every 60 s, a file checkpoint every 60 s, and one snapshot taken only when a reclaim notice arrives. We checked that restored training continues exactly, measured what each protection costs while the job runs, and reclaimed the job in a drill to measure the work lost and the GPU-seconds needed to finish. A trainer compiled with torch.compile [4] tested the case where starting again is expensive.

The answer is mixed, and we report it that way. Every restore was exact. As a periodic protection, snapshots lost to file checkpoints on every measure we took, because a snapshot copies PyTorch's whole GPU memory reserve. A single snapshot taken on a reclaim notice lost no work at all. Snapshots also won when restarting is expensive, as with a compiled model. The direction that follows for the product is a snapshot on notice on top of file checkpoints, not snapshots in place of them.

2 Experimental setup

Platform path. Both hosts ran the Nodus node runtime's guarded runc executor on containerd 2.4.1 and runc 1.5.2, with CRIU 4.3 [5] and NVIDIA cuda-checkpoint [6], the same NVIDIA driver and GPU persistence mode on. The trainer and cuda-checkpoint used the CUDA 13.0 forward-compatible user-mode driver on top of that driver. A research test of the platform's runtime package, which is not merged, drove every run. It starts the trainer as a container through the executor, takes snapshots through the executor's checkpoint call, restores through the executor with a snapshot as the container's checkpoint, and reclaims by killing the container. The two hosts had the same CPU model, kernel and CPU flags, which the executor checks before a restore, and different GPUs.

Workload. The trainer is a character-level GPT with 12 layers, width 768 and 12 heads, 85.35 million parameters, trained with AdamW [7] and bf16 autocast on blocks of 256 characters in batches of 32, on PyTorch 2.7.1. Its 1.1 MB of text sits in GPU memory, batches come from a seeded CPU generator and dropout from the CUDA generator, and PyTorch runs in deterministic mode, so every run of the same steps produces the same bits. The job is 1800 steps of about 0.23 s each, 418.4 s uninterrupted. PyTorch's caching allocator reserved 7.14 GB of GPU memory, of which 1.46 GB is live training state, the parameters, gradients and optimizer state.

Two periodic protections. In the snapshot arm the trainer has no checkpoint code. Every 60 s after its first step the test took a snapshot that leaves the process running, and a copier sent each finished snapshot to the other host's disk. In the file arm the trainer saved its model, optimizer, step and both random generators every 60 s with torch.save to a temporary file, synced it to disk and renamed it, a 1.02 GB file, and the copier sent each one to the other host. Both copies were one stream over the hosts' private network. A protection point counts only once its copy is whole on the recovery host.

Drill and checks. The two arms ran at the same time on opposite hosts, and the test killed each container 270 s after its first step. The recovery host then started from the newest point whose copy was whole before the kill. It restored the snapshot or resumed the checkpoint, with the page cache dropped first, and trained to step 1800. The notice arm took one snapshot that stops the process 150 s after the first step, copied it to the other host and

restored it there. Each run's loss bits at every step, parameter bit-sums every 10 steps and SHA-256 digests at steps 1000 and 1800 were compared with an uninterrupted reference run.

Table 1: The runs. Host A and host B are the two GPU hosts. Hosts name where the protection was taken, then where it was recovered, always the other host. The reference ran 1800 steps uninterrupted on each host, and the resume without random state used the file arm's checkpoint.

Runs	Hosts	Protection	Reclaim
Reference	A and B	None	None
Periodic snapshots	A, then B	Snapshot every 60 s	Killed at 270 s
Periodic file checkpoints	B, then A	Checkpoint every 60 s	Killed at 270 s
Snapshot on notice	A to B, B to A	One stopping snapshot	Notice at 150 s
Resume without random state	A	File checkpoint	None
torch.compile, 600 steps	B, then A	Snapshot at 60 s, checkpoints every 45 s	None

3 A restored snapshot continues bit for bit

Table 2: Each run's steps against the uninterrupted reference on host A, or for the compiled trainer against its own uninterrupted run. Loss bits are compared at every step and parameter bit-sums every 10 steps. The digests are SHA-256 of the parameters, the optimizer state and the random number state.

Run against its reference	Steps	Loss bits differ	Bit-sums differ	SHA-256
Reference on host B	1 to 1800	0	0 of 180	Equal at 1000, 1800
Periodic snapshot, A to B	668 to 1800	0	0 of 114	Equal at 1000, 1800
Notice snapshot, A to B	653 to 1800	0	0 of 115	Equal at 1000, 1800
Notice snapshot, B to A	661 to 1800	0	0 of 114	Equal at 1000, 1800
Compiled, snapshot restored	429 to 600	0	0 of 18	Equal at 600
File checkpoint resumed	1023 to 1800	0	0 of 78	Equal at 1800
Compiled, checkpoint resumed	330 to 600	0	0 of 28	Equal at 600
Resumed without random state	1023 to 1800	778 of 778	78 of 78	Differs at 1800

Exact. All 4 snapshot restores continued the reference trajectory bit for bit on the other host's GPU, with no checkpoint code in the trainer. A snapshot resumes the step that was running when it froze. The periodic snapshot froze during step 668, and the restored process's first line is step 668 with the reference's loss bits. The restored process prints none of the trainer's start-up lines, because it is the same process continuing. Training kept its speed after a snapshot that leaves it running, 0.2297 s per step before against 0.2289 s after. The uninterrupted runs on the two hosts were also identical to each other, so a restore on the other GPU can be checked against either.

A file checkpoint is exact only with care. The careful checkpoint, which saves both random generators, resumed exactly as well. Getting it there took one fix during the experiment, because torch.load had moved the saved generator states to the GPU and the generators refused them. The same checkpoint resumed without its random state, as resume code that saves only the model and optimizer would do, diverged at its first step and never rejoined. Every one of its 778 steps had different loss bits, and its final loss was 1.1823 against 1.2216. A snapshot needs nothing from the trainer except determinism, which a file resume needs too.

4 What a snapshot costs while training runs

Why a snapshot is large. A snapshot holds the process's GPU memory, which cuda-checkpoint moves to host memory, together with the process's own memory. That is PyTorch's whole 7.14 GB reserve, not the 1.46 GB of live state. Each snapshot was 8.53 GB,

Table 3: The cost of one protection point of the eager trainer. A snapshot's stall is how much longer than the median step the step that spans the snapshot took. The lag runs from the start of the snapshot or the save to the copy being whole on the other host.

	GPU snapshot	File checkpoint
Size	8.53 GB	1.02 GB
Training stopped, median	12.9 s, 12.6 to 14.5 s	1.36 s, 1.32 to 1.63 s
Copy to the other host	15.0 to 15.3 s	1.88 to 2.00 s
Lag to a whole copy, median	31.9 s	3.4 s
Slowdown over the first 270 s at a 60 s cadence	24.3%	2.4%

within 1% of the reserve plus the process's 1.42 GB of resident host memory, and 8.3 times the size of the checkpoint file, and 8.51 GB of it is a single page image.

Why a snapshot stalls. CUDA is locked from the moment cuda-checkpoint freezes the process until its state is back on the device, so training makes no progress during a snapshot. The median stall was 12.9 s, 9.5 times the 1.36 s of a torch.save, and the snapshot call returned 0.18 to 4.77 s after training had already resumed. Over the first 270 s, four snapshots left the snapshot arm at step 945, where the reference had passed it at 217.1 s, 24.3% slower. Four checkpoints left the file arm at step 1145, where the reference had passed it at 263.6 s, 2.4% slower. The stall per snapshot is fixed, so a 30-minute cadence would cost 0.7% in stalls, but a longer cadence loses more work at each reclaim.

5 Reclaim drill

Table 4: The reclaim drill. Each arm was killed 270 s after its first step and recovered on the other host from its newest protection point whose copy was whole before the kill. The expected lost work replays a kill at every quarter second against the measured timelines. GPU-seconds to finish count the protected run up to the kill and the recovery run from its start to step 1800, against 422.5 s for the uninterrupted job.

	GPU snapshots	File checkpoints
Step reached at the kill	945	1145
Protection used	Taken at 180 s, step 667	Saved at 240 s, step 1022
Lost work	278 steps, 63.8 s	123 steps, 28.3 s
Expected lost work	184.8 steps, 42.4 s	128.7 steps, 29.5 s
Recovery start	6.98 s, restore	0.32 s, create
Time to the first new step	7.24 s	4.77 s
Time to regain the kill's step	70.4 s	32.6 s
GPU-seconds to finish	544.1 s, +28.8%	459.3 s, +8.7%

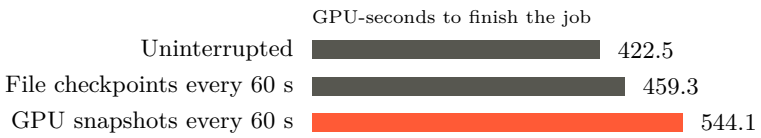


Figure 1: GPU-seconds to finish the 1800-step job through one reclaim at 270 s, lower is better.

Reading. The snapshot arm lost more than twice the work. Its fourth snapshot, taken at 240 s, was still being copied at the kill, with 6.82 of its 8.53 GB on the other host, so recovery used the third, taken at 180 s. The checkpoint written at 240 s had been whole on the other host 3.4 s after the save began. That is the protection lag at work: a snapshot was safe 31.9 s after it started, a checkpoint after 3.4 s. Replaying every kill instant against the measured timelines gives the same order, 42.4 s of expected lost work against 29.5 s. The restore also took longer to reach its first new step, 7.24 s against 4.77 s, because it reads the 8.53 GB snapshot back from disk, while this small trainer starts from scratch in 3.6 s.

Through one reclaim, the snapshot arm needed 544.1 GPU-seconds to finish, against 459.3 s for the file arm and 422.5 s uninterrupted.

6 One snapshot on a reclaim notice

Table 5: One stopping snapshot taken 150 s after the first step, as on a reclaim notice, copied to the other host and restored there with the page cache dropped. Freeze to the first new step counts the snapshot, the copy and the restore, without the test scripts' waits between them.

	Host A to host B	Host B to host A
Last step before the freeze, first step after the restore	652, 653	660, 661
Lost work	0 steps	0 steps
Snapshot	21.4 s, 8.53 GB	20.0 s, 8.53 GB
Copy to the other host	15.3 s	17.1 s
Restore to the first new step	7.0 s	7.1 s
Freeze to the first new step	43.7 s	44.2 s

Nothing lost. A snapshot that stops the process captures it where it is, so the restored process ran the very next step, and neither direction lost or repeated a step. A notice has to cover the snapshot and the copy, 36.7 and 37.0 s here, so a notice of about 37 s is enough for this job, and the restore can follow on the other host after the node is gone. The first new step came 44 s after the freeze, counting only the snapshot, the copy and the restore. The test scripts' waits between those steps made it 72.1 to 73.3 s by the wall clock, 28.0 to 29.6 s that a node starting the copy and the restore at once would not spend. Starting the copy as soon as the dump finishes, rather than when the snapshot call returns, would shorten it further, and we did not measure that. The stopping snapshot took 20.0 to 21.4 s, longer than the 12.9 s stall of a snapshot that leaves the process running, and we did not instrument why.

7 When restarting is expensive: torch.compile

Table 6: The compiled trainer, 600 steps. The snapshot was taken at 60 s and restored on the other host, the checkpoint was written at 45 s and resumed there, each with the page cache dropped. Time to the first new step runs from the start of the restore or the resume. One thread means TORCHINDUCTOR_COMPILE_THREADS=1. Bit-identical means against the uninterrupted compiled run.

	Snapshot restore	File resume	File resume, one thread
Time to the first new step	6.57 s	28.1 s	32.6 s
Ready to first step, compiling	None	23.0 s	27.4 s
Bit-identical	Yes	Yes	Yes

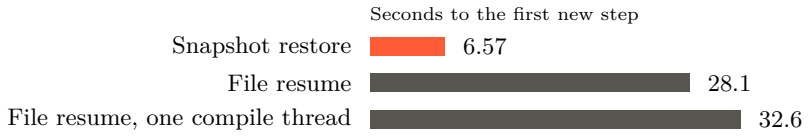


Figure 2: Seconds from the start of the recovery to the compiled trainer's first new step, lower is better.

The restore skips the warm-up. Compiling makes the trainer faster, 0.137 s per step against 0.23 s eager, but its cold start took 29.1 s to the first step against 3.6 s. A file resume pays the compile again. It reached its first new step 28.1 s after it began, 23.0 s of that between the trainer being ready and its first step. The snapshot restore reached its first new step in 6.57 s, because the restored process already holds its compiled code. Even with each

copy fetched only after the reclaim, 7.82 GB in 14.7 s against 1.02 GB in 1.88 s, the restore would reach its first step in 21.2 s against 30.0 s. The compiled snapshot was 7.81 GB and stopped training for 13.3 s, and all three recoveries were bit-identical to the uninterrupted compiled run. The snapshot and the checkpoint come from different points of the run, step 429 against step 330, so this comparison is about the time to the first step, not about lost work.

A process holding an NVML handle blocks the snapshot. The first snapshot of the compiled trainer failed in CRIU. Inductor's default compile pool had started 30 worker processes, and the first of them that CRIU reached had loaded NVML, NVIDIA's management library, and mapped the GPU device /dev/nvidia0. cuda-checkpoint did not release that mapping, so CRIU stopped with "Can't handle non-regular mapping". With TORCHINDUCTOR_COMPILE_THREADS=1, which compiles inside the training process without a pool and was the only change, the snapshot worked, and a file resume with that setting takes 4.5 s longer to compile. We expect any process in the container that holds an NVML handle, such as a GPU monitoring agent, to block a snapshot the same way, but we observed it only for the compile workers. A product has to detect this before it promises a snapshot.

8 What reclaims cost at a given rate

Table 7: A cost model computed from the measured numbers, not measured. With reclaims at a given rate, a periodic protection costs its stall once per period, plus, at each reclaim, half a period of lost work, the protection lag and the time to the first new step [8] [9]. Each periodic protection is shown at its best period, the square root of twice its stall over the reclaim rate. A snapshot on notice costs only its 44 s from the freeze to the first new step at each reclaim.

Reclaims per hour	Periodic GPU snapshots	Periodic file checkpoints	One snapshot on notice
0.25	4.5%, every 610 s	1.4%, every 198 s	0.3%
1	9.6%, every 305 s	3.0%, every 99 s	1.2%
4	21.3%, every 153 s	6.4%, every 50 s	4.9%

Reading. The model uses the measured stalls, lags and times to the first new step, 12.9 s, 31.9 s and 7.24 s for snapshots and 1.36 s, 3.4 s and 4.77 s for checkpoints. File checkpoints come out ahead of periodic snapshots at every rate, 3.0% against 9.6% at one reclaim per hour. A snapshot on notice is cheapest, 1.2% at one reclaim per hour, as long as every reclaim comes with a notice of about 37 s. A job with no checkpoint code restarts from zero at each reclaim. At one reclaim per hour, a one-hour job then expects to run 72% longer and a four-hour job to take 53.6 h. For such a job a snapshot is the only protection that needs no change to the job, and even periodic snapshots bring the cost down to the 9.6% above.

9 What we recommend for reclaim recovery

The measurements point to one design for the reclaim recovery of Nodus Jobs and TrainingJobs, which we recommended and which has since shipped for GPU Jobs as a Beta feature behind the per-organization flag node.gpu_snapshot_jobs, through PRs #728 and #729: a notice-triggered snapshot tier on top of file checkpoints. The cadence rule of the second follow-up below, which counts only the reclaims a notice does not cover, has since shipped too, through PR #756, behind the per-organization Beta flag sched.cadence_unannounced. File checkpoints stay the recovery state, at the job's own cadence. When a reclaim comes with a notice, either a preemption notice from the capacity's owner or a reclaim that Nodus starts itself to drain a node or make room for reserved work, the node takes one stopping snapshot through the executor and uploads it. The replacement attempt restores from that snapshot and falls back to the newest file checkpoint, with a typed reason, if the snapshot does not match the new host, arrives too slowly or fails to restore. Periodic snapshots are offered only to jobs that have no checkpoint code, at the period the cost model picks. Before promising a snapshot, the node checks that no process in the attempt maps the GPU device

outside a CUDA context, as the compile workers did. The file checkpoint keeps the job's contract, and the snapshot only adds a faster path when there is time to take it.

10 Scope

Each arm of the drill ran once and was killed at one point of its protection cycle, 30 s after its fourth protection point, and the replay of every kill instant rests on only 3 or 4 protection points. There was one model size, whose snapshot follows PyTorch's 7.14 GB reserve. Larger models, longer sequences, multi-GPU jobs and trimming the allocator's free reserve before a snapshot were not measured. The copies went host to host over the private network in one stream at about 0.56 GB/s, not through object storage, because the uploader for fleet snapshots did not exist yet when these drills ran. It has since shipped, through PRs #681 and #686, with restores on another node behind the per-organization Beta flag node.gpu_snapshot_fleet. The reclaim was a kill of the container on a host that stayed up, and the recovery host already had the image, so a fresh node pulling the image was not modelled. Restores onto the other GPU used the CUDA 13.0 forward-compatible user-mode driver, and a restore with the host's own driver was not tried. The trainer had no data loader worker processes, no open data files, no NCCL and one GPU. The two drill arms ran at the same time in opposite directions, so each host's training shared its network and disk with the other arm's incoming copies. The cost model is computed, not measured. All 17 runs come from one session on two hosts, and no independent verifier has rerun them.

11 Related work

Periodic checkpointing and its best interval are classic. Young [8] gave the first-order optimum interval, the square root of twice the checkpoint cost times the mean time between failures, and Daly [9] refined it, which is the model used above. For deep learning, CheckFreq [10] tunes the checkpoint frequency at run time and splits each checkpoint into a fast copy of the state and a background write, which shrinks the kind of stall that made periodic snapshots expensive here. Gemini [11] keeps checkpoints of distributed training in the CPU memory of other machines, so that recovery from a failure does not wait for remote storage. These systems save the training state from inside the framework, so the job has to take part.

Transparent checkpointing saves a whole process without the program's help. DMTCP [12] does so for distributed user-space programs, and CRAC [13] extends transparent checkpoint and restart to CUDA programs. CRIU [5] checkpoints Linux processes and containers, NVIDIA's cuda-checkpoint [6] moves a process's GPU state into host memory so that CRIU can save it, and CRIUgpu [14] combines the two for GPU workloads. Singularity [15] uses transparent checkpointing to preempt and migrate deep learning jobs across a fleet. Our reports 033 and 034 measured the same stack restoring inference engines. What this report adds is a measurement of a GPU memory snapshot against the file checkpoint for one training job through the Nodus executor: its exactness, its stall, its lost work in a reclaim drill and the case of a reclaim notice.

12 Follow-up 1, 8 November 2026: releasing the allocator reserve before each snapshot

Question. A snapshot is large because it holds PyTorch's whole 7.14 GB memory reserve, of which 1.46 GB is live state. In a follow-up on two new GPU hosts, the trainer released the free part of that reserve in its own process at a step boundary just before each snapshot, with torch.cuda.synchronize and torch.cuda.empty_cache, and then made no CUDA call until the snapshot had frozen and thawed it. The test asked for each snapshot through a small control file: it wrote a request, the trainer answered at its next step boundary after the release, and the snapshot was taken while the trainer waited. We also tried PyTorch's expandable segments allocator setting alone and together with the release, and reran the file-checkpoint arm in the same session. The trainer, the job, the 60 s cadence and the kill at 270 s were as above, on the same stack, with the runtime test from a later commit of the same unmerged research branch. The new hosts ran at the first hosts' speed, 0.2263 and

0.2258 s per step against 0.2260 s over the first 300 steps, and their 300-step runs matched the reference bit for bit.

Table 8: Each protection every 60 s on the drill runs, killed at 270 s. The untrimmed row is the arm measured above. Reserve is PyTorch's GPU memory reserve when the snapshot was taken. Stall and lag are medians, the lag from the request to the copy being whole on the other host. Slowdown is measured against each run's own median step, which puts the untrimmed arm at 24.6% where the measurement against the reference run above gives 24.3%.

	Snapshot	Reserve	Stall	Slowdown	Lag
Untrimmed	8.53 GB	7.14 GB	12.9 s	24.6%	31.9 s
Release	3.69 GB	2.28 to 2.43 GB	9.1 s	17.1%	22.9 s
Expandable segments	8.30 GB	6.89 GB	12.4 s	23.9%	41.0 s
Both	3.45 GB	2.02 to 2.15 GB	9.0 s	18.4%	20.1 s
File checkpoint, rerun	1.02 GB	None	1.44 s	2.6%	4.9 s

Table 9: The reclaim drill of each variant, killed 270 s after the first step and recovered on the other host from the newest copy that was whole before the kill. Expected lost work replays a kill at every quarter second against each run's measured timeline. GPU-seconds to finish are against 422.0 s, the reference run's 418.4 s from its first step to its end plus a 3.6 s start, which puts the untrimmed arm at +28.9% where the measurement against the reference run's measured 422.5 s above gives +28.8%.

	Untrimmed	Release	Expandable	Both	File, rerun
Step at the kill	945	1000	951	988	1144
Lost in the drill, steps	278	68	282	94	122
Expected lost, steps	184.8	178.1	213.5	156.4	134.5
First new step	7.24 s	3.74 s	7.66 s	3.59 s	4.90 s
Back at the kill's step	70.4 s	19.0 s	72.1 s	24.9 s	32.8 s
GPU-seconds to finish	+28.9%	+13.9%	+29.5%	+16.5%	+9.6%
Bit-identical	Yes	Yes	Yes	Yes	Yes

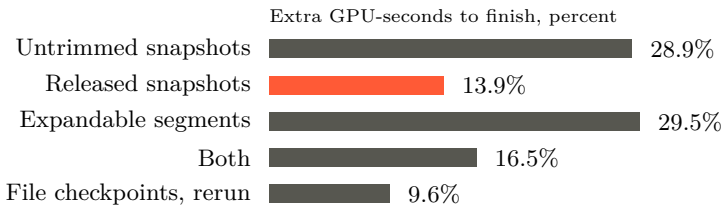


Figure 3: Extra GPU-seconds to finish the 1800-step job through one reclaim at 270 s, in percent over 422.0 s, lower is better.

Smaller and faster to restore, still behind file checkpoints. Releasing the reserve left 2.28 to 2.43 GB reserved, not the 1.46 GB of live state, because segments that still hold live tensors stay mapped, and the snapshot fell from 8.53 GB to 3.69 GB. The release itself took 74 to 88 ms. The restore reached its first new step in 3.74 s against 7.24 s, faster than the file resume's 4.90 s, and in the drill the released arm lost 68 steps against 122 for the file checkpoint. That drill was lucky: the newest snapshot's copy had become whole 2.1 s before the kill. Replaying every kill instant gives 178.1 steps of expected lost work against 134.5 for file checkpoints, because each snapshot still stopped training for 9.1 s at the median against 1.44 s for a torch.save, 6.3 times as long, and still took 22.9 s to be safe on the other host against 4.9 s. Through one reclaim the released arm needed 13.9% more GPU-seconds than the uninterrupted job, against 9.6% for the file arm rerun in the same session. Expandable segments alone changed little, 8.30 GB and a 12.4 s stall, and together with the release they gave 3.45 GB against 3.69 GB at a 9.0 s stall. All 5 snapshot restores of this session, each on the other host, and the file resume continued bit for bit, with equal

SHA-256 digests at steps 1000 and 1800, and the training that continued on each original host after its snapshots matched the reference up to the kill.

Where the stall goes. In each released snapshot the trainer was held from its answer until the freeze thawed it, 8.3 s at the median, and the first step after the thaw took 0.64 to 0.76 s longer than usual. A separate run on host A timed cuda-checkpoint's two moves on the live trainer: 1.12 to 1.16 s to move the GPU state into host memory after a release, against 2.82 to 2.91 s with the full reserve, and 0.54 to 0.55 s to move it back, against 1.22 s. CRIU's own log puts the dump of a released image at 1.84 to 1.95 s, against 3.94 to 3.97 s for the 8.30 GB images. In that separate run, where the state was back on the GPU before the trainer went on, the next step took the usual 0.227 to 0.228 s, so regrowing the released reserve costs no measurable time, and the slow first step after a real snapshot matches moving the state back to the GPU after the thaw. What is left of each hold after the move out and the dump is about 5.3 s at the median, 4.2 to 8.9 s per snapshot, and about 4.3 to 4.9 s without the release. That remainder lies in the rest of the node runtime's checkpoint path, the pause, containerd's checkpoint call, the start-up of runc and CRIU and the resume, which no allocator setting touches. It is a remainder, not a measurement inside the checkpoint path. The trainer also thawed 0.8 to 2.9 s before each checkpoint call returned.

A snapshot on notice gets better. With the release, one held stopping snapshot taken 150 s in, as on a reclaim notice, again lost no steps in either direction. Its 3.92 GB took 11.2 and 15.6 s from the request to the end of the snapshot and 12.6 and 13.6 s to copy, so it needs a notice of 23.8 to 29.2 s instead of about 37 s, and the first new step came 27.6 to 33.1 s after the freeze instead of 43.7 to 44.2 s, counting only the snapshot, the copy and the restore. The test scripts' waits made it 54.8 to 56.7 s by the wall clock. Both restores were bit-identical. The cost model above, with these numbers, keeps file checkpoints ahead of periodic snapshots at every rate, 3.1% against 7.8% for released snapshots at one reclaim per hour, 1.5% against 3.7% at 0.25 and 6.8% against 17.2% at 4.

What changes in the recommendation. We recommend, and it is not built, that the snapshot on notice release the reserve first, at a step boundary, through a take point for Jobs and TrainingJobs that reuses the state-file protocol of Nodus Functions. The node writes a take request, the worker releases the cache at its next step boundary, as the Functions SDK already does before each offer, answers with its memory figures and makes no CUDA call until it is thawed or told the take is over, and a restored copy continues the same way. Expandable segments should not be set, because they change the customer's allocator for little gain. Periodic protection stays on file checkpoints, and the roughly 5 s left in the node runtime's checkpoint path is the next thing to profile, before any more allocator work.

Scope of the follow-up. There was one model and one job, whose fragmentation after the release may differ on other models, and one drill per variant at one point of its cycle, with the replay resting on 3 or 4 protection points each. The remainder in the checkpoint path is what is left after subtracting measured parts, not instrumented, and the two moves come from a separate run with the default allocator. The paired drills ran at the same time and copied in both directions at once. A step-boundary hook that needs no change to the trainer was not built or measured, because this trainer checked for requests at the end of its own loop. No independent verifier has rerun these measurements.

13 Follow-up 2, 9 November 2026: the file-checkpoint cadence once every reclaim notice gets a snapshot

Question. The sections above found that one stopping snapshot taken on a reclaim notice loses no steps, and recommended it on top of file checkpoints. The snapshot on notice for GPU Jobs is now in the product, behind the per-org Beta flag node.gpu_snapshot_jobs (PRs #728 and #729, merged). A job stopped this way loses nothing to a noticed reclaim, so its file checkpoints only protect it against failures that come without notice. The product still sets a Job's file-checkpoint cadence from every reclaim, noticed or not, with the classic checkpoint-interval formula of Young and Daly [8] [9], the square root of twice the checkpoint cost times the mean time between failures. This follow-up asks how much more training one

GPU gets done on reclaimable capacity when the cadence counts only the failures a notice does not cover. That cadence rule has since shipped in the Nodus platform behind the per-organization Beta flag sched.cadence_unannounced, off by default, through PR #756, merged on 9 November 2026.

Setup. One GPU trained a 302.7 million parameter GPT, the character-level trainer of the sections above scaled to 24 layers of width 1024, compiled with torch.compile and deterministic, at 0.2545 s per step. A file checkpoint of its model, optimizer, step and random generators was 3.63 GB and stalled training for 4.79 s on average, 4.54 to 5.19 s across runs. A cold start on a replacement copies the newest checkpoint file, drops the page cache, loads it and compiles again, and took 55.1 s on average from the kill to the first new step. The stop snapshot used cuda-checkpoint and CRIU 4.3 on the trainer as a bare host process, not through the product's runc executor, and was restored on the same GPU. Before each snapshot the trainer released PyTorch's free reserve at a step boundary, as in follow-up 1, and the image was 5.90 GB at the median.

Events and compressed time. Each run lasted 1500 s from its first step and replayed one of 3 seeded schedules. Noticed reclaims came every 300 s on average, each with a lead of 30 to 120 s before the machine was taken, and unannounced kills came every 1800 s on average. Time was compressed 12 times. The gaps stand for one noticed reclaim an hour and one unannounced loss every 6 hours, but every cost is the GPU's real cost, the 4.79 s save, the snapshot cycle and the 55.1 s cold start. Each cost therefore weighs 12 times more against the gaps than it would at real rates, so the absolute goodputs are those of a very hostile fleet. The differences between the arms and the cost of each event carry over, and the model below rescales them to real rates. Goodput is the steps a run had trained at the end of its window over the 5834 steps an uninterrupted run trained in the same 1500 s.

Four arms. A is today's product without the flag. It writes a file checkpoint every 60 s, and on a notice an urgent one, after which the process keeps training until the machine is taken and the replacement starts cold from the newest file. B adds the snapshot on notice: after the urgent checkpoint the process stops in one snapshot and continues from it. C is B with files every 131 s, the classic interval for the unannounced rate alone, from the 4.77 s checkpoint measured before any arm ran and the 1800 s mean gap. D takes the snapshot on notice and writes no files. After an unannounced kill, B and C restore the paired snapshot while no newer file exists and otherwise start cold from the newest file, and D restores its newest snapshot. A's 60 s is close to the 50 s the same rule gives when it counts every failure.

Table 10: Goodput, the steps trained at the end of the 1500 s window over the uninterrupted run's 5834, for each seed and on average, and the mean steps per GPU-second against 3.889 uninterrupted. Seed 1 has no unannounced kill, and seeds 2 and 3 have one each.

Arm	Seed 1	Seed 2	Seed 3	Mean	Steps per GPU-second
A, files every 60 s, urgent file on notice	0.713	0.815	0.644	0.724	2.816
B, A plus the snapshot on notice	0.869	0.838	0.835	0.848	3.296
C, snapshot on notice, files every 131 s	0.904	0.856	0.820	0.860	3.344
D, snapshot on notice, no files	0.949	0.738	0.537	0.742	2.884

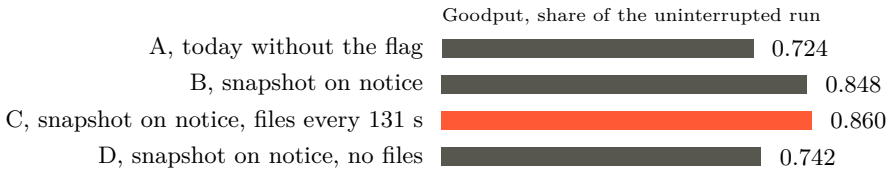


Figure 4: Mean goodput over the 3 seeds, the share of the uninterrupted run's steps, higher is better.

The snapshot on notice is the big win. B reached 0.848 of the uninterrupted run against 0.724 for A, 12.4 points more on average and +15.6, +2.4 and +19.1 by seed, or 3.296

Table 11: What each event cost, pooled over the 3 seeds. A paid for 9 reclaims from its 13 notices, because 3 notices fell on a machine an earlier notice had already doomed and 1 reclaim fell after the window. In each snapshot arm the 13th notice was snapshotted, but the window ended before its restore. A's first new step after a reclaim is the mean from the kill, the snapshot arms' the median from the notice.

	A	B	C	D
Noticed reclaims	9 reclaims	12 restored	12 restored	12 restored
Steps lost per noticed reclaim	22 to 179, mean 101.8	0	0	0
First new step after a noticed reclaim	54.9 s, cold start	22.4 s	22.5 s	16.4 s
Steps lost to the 2 unannounced kills	125, 147	143, 3	304, 300	1377, 2384
Periodic file-checkpoint stall, share of the window	5.84%	6.56%	2.67%	None

steps per GPU-second against 2.816. The snapshot arms lost 0 steps on all 36 notices they restored, 12 per arm, and every snapshot was whole on the replacement's disk at least 15.1 s before its reclaim. From the notice to the first new step took 22.5 s at the median in B and C, with the 4.72 s urgent checkpoint, and 16.4 s in D. A lost 22 to 179 steps at each of its 9 reclaims, 101.8 on average, and then paid a cold start. The urgent checkpoint without a stop still loses work, because the process trains on after it until its machine is gone, and that training is lost. In seed 2, A's notices at 937.3 s and 954.8 s fell on one machine, so A paid one reclaim for both, which is why B gained only 2.4 points there.

Setting the cadence from unannounced failures adds a smaller gain. C reached 0.860, 1.2 points over B on average, with +3.5, +1.7 and -1.6 by seed. The longer cadence cut the periodic file-checkpoint stall from 6.56% to 2.67% of the window, 3.9 points, and gave 2.6 points back in work redone after the unannounced kills, where C lost 304 and 300 steps against 143 and 3 for B. The gain is noisy. Each arm met only 2 unannounced kills, and in seed 3 B's kill came 3 steps after a file checkpoint while C's came 300 steps after one. The model below, calibrated with the measured costs, expects +3.0 points at this failure mix, inside the per-seed range of -1.6 to +3.5.

Without file checkpoints, an unannounced kill is catastrophic. D reached 0.949 in seed 1, which has no unannounced kill, the best of any run, but 0.738 and 0.537 in the two seeds with one. Each unannounced kill sent D back to its last notice snapshot, 1377 and 2384 steps earlier, and D's mean of 0.742 fell below B's. A snapshot on notice covers only the failures that come with a notice, so file checkpoints stay, at the cadence the unannounced failures set.

Every step exact. Every step line any arm logged, 61,588 with replays included, has the uninterrupted run's loss bits, and all 569 fingerprints of the parameters, the optimizer state and the random number state match, through restores from files and from snapshots.

Table 12: A model calibrated with the measured costs, not a measurement. The goodput lost to file checkpoints and unannounced kills is the checkpoint stall over the cadence plus the unannounced rate times the expected cost of one unannounced kill, with notices every 300 s, a 4.79 s checkpoint, a 55.1 s cold start and a 4.8 s restore of a paired snapshot. The cost of the notices is the same for B and C and cancels. Times are compressed seconds.

Unannounced kill every	Lost, B, files every 60 s	Lost, C, files every 131 s	C against B
300 s	33.0%	36.3%	-3.4 points
450 s	24.6%	25.4%	-0.8 points
600 s	20.5%	20.0%	+0.5 points
900 s	16.3%	14.5%	+1.8 points
1800 s, the design rate	12.1%	9.1%	+3.0 points
3600 s	10.1%	6.4%	+3.7 points
7200 s	9.0%	5.0%	+4.0 points

Break-even and product scale, from the model. With the cadence held at 131 s, C stops beating B once unannounced kills come more often than about one per 534 s of compressed time, 3.4 times the design rate of one per 1800 s. Plain Young and Daly costs put that

break-even at 821 s, the product of the two cadences over twice the checkpoint cost. At real rates, with the product's own defaults of a 1 minute write, the 5 minute to 2 hour clamp and the 10% duty floor, and with one noticed reclaim an hour and one unannounced loss a day, the model puts the cadence at 644 s today and at 3220 s when it counts only unannounced losses. File checkpoints plus redone work then fall from 9.7% to 3.7% of GPU time. Both figures assume the snapshot on notice, and none of these numbers is a measurement.

What changes in the product. We recommend that a Job which will get the snapshot on notice have its cadence set from the failures a notice does not cover, the hazard of heartbeat loss plus the reclaim hazard times the share of reclaims whose notice came with enough lead and whose snapshot was restored. That share should start at 0 and grow only from measured outcomes, so nothing changes without evidence, and the unannounced hazard should stay measured, not zero, because the relaxed cadence loses once unannounced failures become frequent. The rule should follow the snapshot path, not notices in general, because an urgent checkpoint without a stop still loses the training done after it. This cadence rule is now in the product behind the per-organization Beta flag sched.cadence_unannounced through PR #756.

Scope of the follow-up. There was one workload, the 302.7 million parameter compiled trainer, on one GPU, and the gain from the cadence scales with the checkpoint cost over the cadence. Time was compressed 12 times, and only the model's real-rate figures use real rates. Each arm met only 2 unannounced kills, so the measured gain of C over B rests on where those kills landed. Copies were local. The snapshot's upload and download was a local copy and sync, 5.1 s for 5.90 GB, and a cold start copied the checkpoint file locally. At the 4.6 s up and 5.4 s down for a 2.30 GB image that a GPU test of PR #729 measured through object storage, each notice would take about 20 s more. The snapshot ran through cuda-checkpoint and CRIU on a bare host process, not the product's runc executor, and the replacement was the same GPU, at once. In seeds 2 and 3 two notices came 17.5 and 4.0 s apart. A paid one reclaim for both, while the snapshot arms took a second snapshot of the restored process, which cost B and C 20.5 to 22.4 s, 1.4 to 1.5 points, and D 16.1 to 16.3 s in each of those seeds. The first A run of seed 1 hit a bug in the test driver that let a reclaim meant for a replaced machine kill the replacement, and scored 0.690. It is kept on record, and A seed 1 was rerun after the fix. No independent verifier has rerun these measurements.

14 Conclusion

A GPU memory snapshot is an exact way to move a training process off a GPU that is being reclaimed. On two GPUs it continued bit for bit in all 4 restores, with no checkpoint code in the trainer. It is not a cheap way to protect a job that already checkpoints. It holds PyTorch's whole memory reserve, 8.53 GB against a 1.02 GB file, and each periodic snapshot stopped training for 12.9 s against 1.36 s, so in a reclaim drill the job needed 28.8% more GPU-seconds to finish against 8.7% with file checkpoints. Snapshots win in two places: a single snapshot on a reclaim notice, which lost no steps and resumed training on the other host 44 s after the freeze, and jobs whose restart is expensive, where a compiled trainer reached its first step in 6.6 s against 28.1 s. For a fleet, that makes snapshots a notice-triggered tier on top of file checkpoints, so that reclaimable and idle GPUs can train with less work lost. Whether this holds for larger models, multi-GPU jobs and copies through object storage is the next measurement. Follow-up 1 found that releasing PyTorch's free reserve before each snapshot halves its size and the restore time and shortens the notice it needs to 23.8 to 29.2 s, which makes the snapshot on notice better, while periodic snapshots still lose to file checkpoints. Follow-up 2 found that once every reclaim notice gets a snapshot, file checkpoints only need to cover the failures that come without notice: on one GPU the snapshot on notice added 12.4 points of goodput, a cadence set from unannounced failures alone added a smaller 1.2 points, and a job with no file checkpoints lost 1377 and 2384 steps to its two unannounced kills.

References

[1] Nodus Research. GPU Memory Snapshots in Containers. Report 033, 2026.

[2] Nodus Research. Several Models per GPU by Swapping GPU Memory Snapshots. Report 034, 2026.

[3] A. Paszke et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. NeurIPS 2019.

[4] J. Ansel et al. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. ASPLOS 2024.

[5] CRIU project. Checkpoint/Restore In Userspace. criu.org.

[6] NVIDIA. cuda-checkpoint: checkpoint and restore of CUDA process state. github.com/NVIDIA/cuda-checkpoint, 2024.

[7] I. Loshchilov, F. Hutter. Decoupled Weight Decay Regularization. ICLR 2019.

[8] J. W. Young. A First Order Approximation to the Optimum Checkpoint Interval. Communications of the ACM, 1974.

[9] J. T. Daly. A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. Future Generation Computer Systems, 2006.

[10] J. Mohan, A. Phanishayee, V. Chidambaram. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. FAST 2021.

[11] Z. Wang et al. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. SOSP 2023.

[12] J. Ansel, K. Arya, G. Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. IPDPS 2009.

[13] T. Jain, G. Cooperman. CRAC: Checkpoint-Restart Architecture for CUDA with Streams and UVM. SC 2020.

[14] R. Stoyanov et al. CRIUgpu: Transparent Checkpointing of GPU-Accelerated Workloads. arXiv preprint arXiv:2502.16631, 2025.

[15] D. Shukla et al. Singularity: Planet-Scale, Preemptive and Elastic Scheduling of AI Workloads. arXiv preprint arXiv:2202.07848, 2022.