Understudy: Putting GPU Fault Stalls to Work

Samarth Narang[1,2] Aamer Jaleel[2] Brandon Reagen[1]
[1]New York University [2]NVIDIA

arXiv:submit/8387313 [cs.AR] 17 Nov 2026

Abstract

Modern GPUs serve a wide range of applications, giving operators a natural opportunity to improve utilization by colocating independently managed applications on a single GPU. In practice, a resident service keeps its working set in GPU memory to meet strict latency targets, while a colocated job often uses demand paging to bring pages in from CPU memory as needed. On two NVIDIA Blackwell GPUs, however, we find that when colocated jobs trigger page faults on the streaming multiprocessors (SMs), the hardware cannot switch contexts until the driver finishes servicing those faults, stalling the resident service even though all of its data is already in GPU memory and most of the GPU's compute units sit idle. Reclaiming that idle capacity is not straightforward because thread blocks waiting on page faults continue to hold their registers, shared memory, and slots on the SMs.

We present Understudy, a GPU scheduling framework that lets colocated applications run during fault service by combining resource sharing with selective state saving. When an application submits a kernel during a colocated job's fault stall, Understudy places its thread blocks into free SM resources and measures how quickly they progress with that capacity. Because freeing additional capacity requires saving stalled blocks from the faulting application and restoring them later, Understudy saves selected stalled blocks only when the predicted speedup for the submitted work outweighs that save and restore cost, letting the driver continue servicing faults and restoring the blocks as resources free up after fault replay. We evaluate Understudy in a cycle level simulator calibrated against hardware measurements and compare it against default scheduling and two hardware baselines that either share free SM resources without saving state or switch out the entire faulting context. Across seven workload pairs, Understudy achieves a geometric mean speedup of 147× over default scheduling for work submitted during a colocated job's page fault stall (finishing at 1.27× its isolated runtime on average), while adding at most 1.6% to the faulting application's completion time.

1 Introduction

Colocating latency-sensitive foreground services with background jobs can improve GPU utilization by using idle capacity [11, 20, 25, 38, 46, 69, 76, 81, 82, 86, 88, 89]. These applications are often developed and deployed independently, with different policies for placing data in GPU memory. A foreground service may keep its working set resident, while a background job migrates data from CPU memory on demand. We find that the background job's page faults can nevertheless prevent the foreground service from running, even when both applications fit in GPU memory and most compute units sit idle.

ISCA '27, Atlanta, Georgia, USA
2027. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

Consider a DLRM job [45] starting alongside a resident Qwen2.5-3B decoding service [85] on an RTX PRO 2000 (see Figure 1). Before the DLRM job arrives, Qwen's decode step produces a token every 25 ms. When DLRM starts, both applications fit within the GPU's 16 GB of memory, yet DLRM faults on 8.4 GiB of embedding tables that begin in CPU memory. During DLRM's 1.2 s fault episode, Qwen completes only 12 decode steps, compared with 48 in isolation. Although Qwen itself never faults, we observe that, across multiple runs, 9 to 11 of these steps take more than twice the usual 25 ms, with the slowest reaching 160 ms. Once DLRM's tables are resident in GPU memory and the two applications continue running together, Qwen's decode latency settles to 29 ms, showing that the fault stalls at the background job's arrival disrupt the foreground service far more than steady state resource contention.

These delays arise because fault handling prevents the scheduler from switching applications. NVIDIA GPUs have supported instruction-level compute preemption since Pascal (GP100) [53], but our measurements on two Blackwell GPUs show that unresolved faults from streaming multiprocessor (SM) execution prevent the hardware from completing a context switch. A fault batch takes a median of 410 μs to service on the RTX PRO 2000, compared with 16 μs for an average decoder kernel. Replay can immediately generate additional faults, sustaining intervals during which the faulting context cannot be preempted. Missing pages in one application therefore delay unrelated work whose data is already resident.

Software can mitigate interference through prefetching or remote mapping (Section 3.2). Prefetching all of DLRM's embedding tables eliminates the fault stalls that delay Qwen, but waiting for the transfer increases the time to complete DLRM's first batch from 0.37 s to 0.54 s. Whether guided by hand or by an automated runtime [8, 65], prefetching requires enough free GPU memory to hold the prefetched pages alongside colocated applications, and any page it fails to predict still triggers a demand fault. Even with 90% of the tables prefetched, some Qwen decode steps still exceed 50 ms, compared with 25 ms in isolation. Remote mapping avoids migration faults by keeping the tables in CPU memory [72], but accessing them over PCIe reduces DLRM's isolated throughput by more than half. We therefore target the hardware scheduler so that one application can execute while the driver services another's page faults, without changes to either application.

Enabling that progress also requires managing resources held by stalled execution. A fault-stalled thread block retains its registers, shared memory, and warp and block slots, so idle compute units do not imply that another block can fit. Prior work shares SM resources among kernels [2, 78], makes faulting code restartable [43, 71], selectively reclaims resources [56, 79], and switches out a block whose warps all stall on page faults so that another block of the same kernel can expose more faults for batching [33]. Colocation needs these capabilities combined across independently scheduled applications: letting one application make progress during another's fault service, isolating its memory requests from the faulting one,

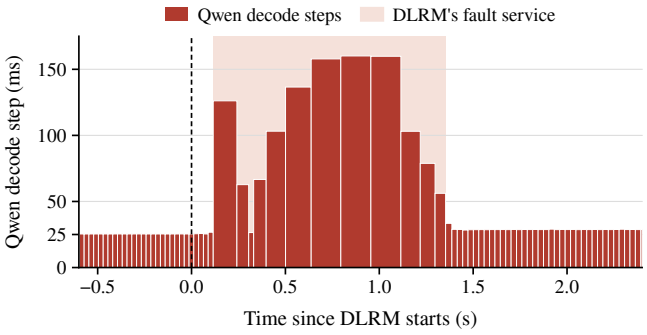


Figure 1: Decode step time of a resident Qwen2.5-3B service colocated with a DLRM job.

and deciding whether additional capacity will repay the cost of saving state. A fault-and-switch scheduler could save the faulting application's entire context and run waiting work during fault service. However, our measurements of the baseline preemption show that state saving and restoration incur a substantial fixed transition delay plus a cost that grows with the state transferred. A full context switch is unnecessary when waiting work fits in unused resources, and excessive when releasing a few stalled blocks provides sufficient capacity.

We present **Understudy**, a GPU scheduling framework that lets colocated applications make progress during page fault service by combining per-application fault isolation with selective state saving. Understudy extends the hardware subcontext support used by NVIDIA's Multi-Process Service (MPS) [51] so that blocks from independently scheduled applications can execute concurrently on the same SMs. Understudy isolates fault stalls in the memory pipeline, allowing one application's memory accesses to proceed while another's requests remain stalled.

To reclaim additional capacity held by stalled blocks, Understudy must preserve their unfinished memory operations while saving their execution state. Preemption mechanisms that drain pending requests before saving must wait for fault service to finish [78], while hardware exception mechanisms track potentially faulting instructions during normal execution to support restart [71]. Understudy instead captures unfinished requests at save time. It stops instruction issue, lets requests already dispatched to the caches complete, and saves the remaining requests with the block's execution state. When the block is restored, these requests reissue before execution resumes, preserving completed work without continuous instruction tracking (Section 4.4).

With this support, Understudy first places arriving blocks in free resources, comparing each kernel's block requirements with the resources available on each SM. If no block fits, it saves eligible stalled blocks to enable admission. Once work is running, it uses observed block completion rates and estimated remaining fault-service time to decide whether additional capacity justifies the cost of saving and restoring more state. It favors blocks that release the required resources with less state movement. When fault service finishes, blocks that remained resident resume, while saved blocks reclaim resources as admitted work completes.

We characterize this fault interference on two Blackwell GPUs and evaluate Understudy using a cycle level simulator built on Accel-Sim [12, 32]. Across seven workload pairs where an application submits work during a neighbor's first touch fault episode, Understudy reduces the arriving work's geometric mean completion time from 187× isolated execution under default scheduling to 1.27× (a 147× speedup) while adding at most 1.6% to the faulting application's completion time in our primary configuration. Compared with fault and switch, Understudy completes the arriving work 1.26× faster and cuts total state movement by 44%. These gains are made possible by isolating fault stalls per application in the memory pipeline and saving stalled blocks that still hold unresolved memory requests, while our ablations show that admitting work into free resources first and deciding online when to save state provide most of the speedup and that saving individual blocks moves 27% less state than saving whole SMs at nearly identical completion times.

This paper makes four contributions:

- We characterize fault-induced scheduling interference on two Blackwell GPUs, showing that unresolved SM faults delay resident applications even when combined working sets fit in device memory.
- We design hardware support for per-application fault isolation and saving stalled blocks by capturing pending requests at save time, enabling execution and resource reclamation during unresolved faults.
- We develop an online scheduling policy that admits work into free resources and weighs the benefit of additional capacity against the cost of saving and restoring state.
- We evaluate Understudy against SM sharing and fault and switch across ablations, repeated submissions, changing kernel sequences, and multiple faulting applications.

2 Background

2.1 Contexts and TSG scheduling

GPUs now accelerate workloads ranging from scientific simulation and bioinformatics to irregular graph analytics, data processing, and machine learning [1, 17, 19, 21, 26, 40, 42, 47]. Each GPU application runs inside a CUDA context that holds its virtual address space and execution state, and submits kernel launches and memory copies through ordered streams. The driver places these commands into hardware queues called channels and groups the channels of an application into a time slice group (TSG) [50]. A hardware runlist scheduler maintains a list of active TSGs and selects which group runs on the compute engine [13–15, 18, 50, 51, 77], allowing that group's kernels to dispatch thread blocks to the SMs.

Each TSG runs for a fixed time quantum before the scheduler switches to the next group in the runlist. When the quantum expires, however, the outgoing group's running blocks must either finish or be preempted before the next group can use their SM resources. Preempting unfinished blocks requires saving their registers, program counters, shared memory, and barrier state to device memory so they can resume later, which means a waiting application experiences both the delay until the scheduler requests a switch and the time needed to save that state.

2.2 Block Placement and Resource Allocation

Once a TSG is running on the compute engine, the hardware work distributor dispatches its kernels' thread blocks, or cooperative thread arrays (CTAs), to the SMs. Each block contains warps of up to 32 threads and can enter an SM only when enough registers, shared memory, warp slots, and block slots are free at the same time; because a block holds all of these resources until it finishes and never releases them when its warps stall on memory accesses, an SM can issue almost no instructions while still lacking the free registers or shared memory needed to admit another block.

For a second application to use the free resources on an SM, the hardware must be able to run blocks from two address spaces at the same time. Volta and later GPUs already support this under MPS by giving each client its own subcontext and page tables within a shared TSG, tagging each client's work with a subcontext identifier so its memory accesses use the right page tables [15, 50].

2.3 Demand Paging and Fault Service

CUDA managed memory lets the CPU and GPU share a single virtual address space while the Unified Virtual Memory (UVM) driver tracks where pages reside and updates their page tables on demand. When a GPU thread accesses a page that has no valid mapping on the device, the hardware records a replayable page fault and parks the memory request [59, 60, 68, 73]. The driver then reads these faults in batches, migrates the missing pages into GPU memory, updates the GPU page tables, and issues a replay so the GPU retries the parked requests and the waiting warps continue execution.

How an engine handles page faults also determines whether another application can run while the driver services those faults. NVIDIA's open source UVM driver states that replayable faults on the SMs follow a fault and stall model that prevents the hardware from preempting the faulting channel until the driver resolves the faults [39, 52]. By contrast, recoverable copy engine faults on the non replayable path follow a fault and switch model: the hardware preempts the faulting TSG so other groups can use the copy engine while the driver resolves the faults, and then marks the group runnable again. Compute SMs have no equivalent mechanism and keep the faulting group on the hardware throughout fault service.

3 Characterizing Fault-Induced Interference

3.1 Scheduling During Fault Service

To measure how one application's demand paging affects a colocated neighbor, we run two processes, A and B, on one GPU where A touches managed memory that starts in CPU memory and faults, while B uses only data already in GPU memory and never faults. We first measure launch delay from the CPU's submission of B's kernel to its first instruction on the GPU. B starts within 30 to 40 μs when A has no kernel running, but waits a median of 77 ms when submitted during A's fault service. Although each fault batch takes a median of 410 μs to service on the RTX PRO 2000, our UVM traces show that A's warps fault again as soon as each replay lets them advance. Because those batches follow back to back, A holds unresolved faults across 87 to 100% of its faulting kernel runtime and leaves B waiting through the whole interval. As soon as A's

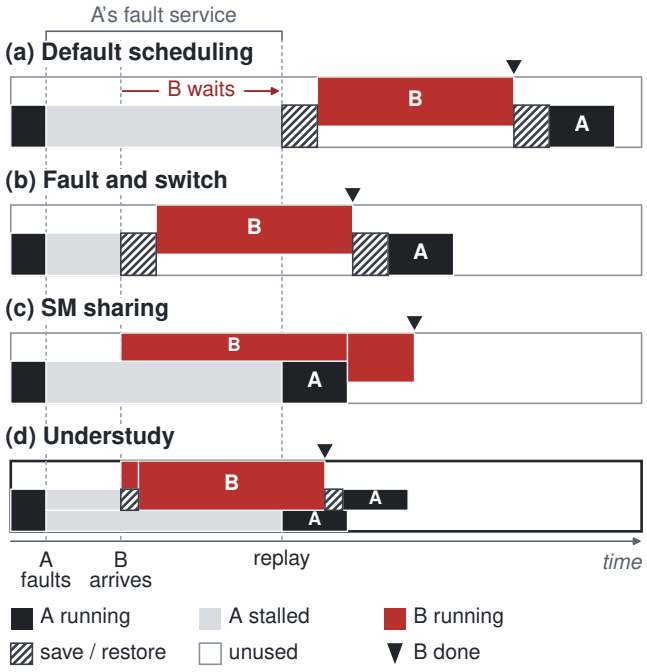


Figure 2: Scheduling B during A's fault service.

faults end, the scheduler preempts A and switches to B even if A's kernel still has compute work left, confirming on both the RTX PRO 4000 and RTX PRO 2000 that unresolved faults rather than kernel execution block the switch.

Second, A's faults also stall B's already running kernels and B's memory copies that do not use the SMs. When B launches first and its threads repeatedly read the GPU cycle counter, consecutive readings stay within 0.3 ms while A's data resides in GPU memory, whereas those gaps stretch to 84 to 92 ms when A's data starts in CPU memory and A faults. Likewise, a 1 MB copy between two resident GPU buffers takes 18 μs alone, but waits 67 to 88 ms when submitted during A's SM fault service even though it requires neither page migration nor SM thread blocks.

We observe the impact of these stalls across nine background workloads colocated with Qwen2.5-3B decoding on the RTX PRO 2000. As long as both applications fit in device memory, the background workloads slow decoding by only 1.1 to 1.7× over a full run, whereas once their combined data exceeds device memory so that pages are repeatedly evicted and faulted back in, we find that DLRM slows decoding by 7.5× and five of the other eight workloads slow it by 14 to 127×.

To rule out fault buffer traffic as the cause, we make A fault from copy engine transfers instead of SM kernels. Copy engine faults normally take the non replayable path that preempts the faulting TSG (Section 2.3), yet even when we route A's copy engine faults through the replayable fault buffer at 3.4 million faults per second, B's kernels keep starting, showing that unresolved SM faults are what keep the hardware from preempting A's compute context. We next modify the open GPU kernel driver to switch compute contexts during SM fault service just as it does for the copy engine.

When our driver requests a runlist switch to B after servicing a batch of A's faults, B stays queued behind A, and forcing the switch by disabling A's channels triggers a context switch timeout and a fatal fault that terminates A. While these tests cover only the driver mechanisms we tried, they indicate that suspending A during SM fault service requires hardware support to save A's execution state and unresolved memory requests while the driver services them.

Figure 2 compares hardware scheduling options when B arrives during A's fault service. While suspending A's entire context (fault and switch in Figure 2(b)) lets B run immediately, it saves all of A's state even when B would fit into the SM resources that A leaves free. Conversely, placing B only into A's unused resources (SM sharing in Figure 2(c)) moves no state, but leaves B blocked or slow whenever A holds most of the SMs. Understudy (Figure 2(d)) combines both by placing B into free resources first and saving selected stalled blocks of A only when the expected reduction in B's completion time outweighs the cost of saving and restoring them.

3.2 Limits of Software Alternatives

We next test whether software techniques can eliminate this interference. In the arrival scenario of Figure 1, having DLRM prefetch 100% of its 8.4 GiB tables at startup eliminates demand faults and keeps every Qwen decode step at or below 33 ms, but delays DLRM's own first batch from 0.37 s to 0.54 s. CUDA treats prefetching as an optional hint [49], and protecting the resident service this way requires every colocated job to know and prefetch all of its pages in advance. When DLRM prefetches only 50% or 90% of its tables, the remaining pages still fault on the SMs and stall 8 to 9 or 2 to 3 decode steps above 50 ms (peaking at 104 ms and 58 ms). When two independent DLRM jobs with 4 GB and 3 GB tables arrive alongside the decoder and only the 4 GB job prefetches, the 3 GB job's faults still push 15 decode steps above 50 ms. Understudy complements prefetching by letting the resident service run through the remaining faults.

Remote mapping instead pins the job's data in CPU memory and sets up its GPU page table entries in advance so the GPU reads it directly over PCIe. While this avoids SM faults and raises Qwen's decode throughput from 4.8 to 24.6 tokens/s when DLRM's tables exceed GPU memory, it cuts DLRM's standalone throughput from 878,000 to 386,000 recommendations/s across its entire run when those tables would otherwise fit on the GPU.

We next test whether MPS (Section 2.2) lets B use the resources left idle while A waits on page faults. Even with MPS enabled, we find that a kernel submitted by B while A is faulting still waits roughly 78 ms before starting, showing that merging applications into a shared TSG under MPS does not prevent A's page faults from stalling B.

3.3 Available Resources During Fault Service

Although A's thread blocks retain their allocated resources while waiting on page faults, those allocations do not necessarily fill the GPU. Across eight faulting workloads on the RTX PRO 2000 (Figure 3), we find that SMs issue only 0.06 to 0.33% of peak instructions during fault stalls, yet the stalled blocks continue to hold 21 to 91% of the warp slots and 24 to 87% of the registers, leaving compute units nearly idle while locking up a widely varying share of each

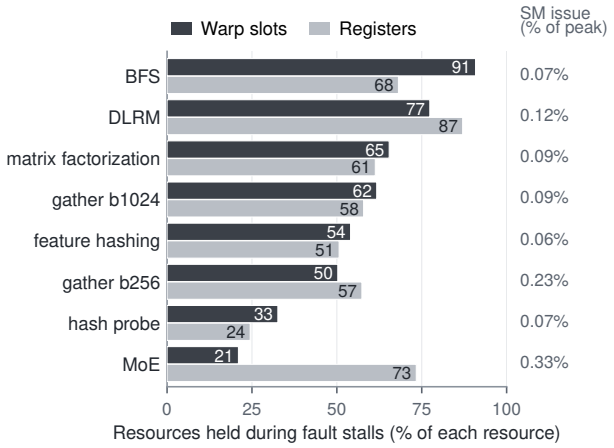


Figure 3: Resources held and SM issue on the RTX PRO 2000, averaged over fault stall intervals with issue below 2%.

SM's capacity. Moreover, whichever resource runs out first limits sharing: in our mixture of experts (MoE) workload, each selected expert runs a cuBLAS [48] kernel compiled with 164 registers per thread, so its fault stalled blocks occupy on average only 21% of the warp slots yet consume 73% of the register file and prevent another application from using the remaining warp slots. Because A's free resources may fit too few of B's blocks for fast progress while saving and restoring A's state carries overhead, the scheduler must decide online whether to run B in place or save stalled blocks for extra room.

4 Understudy

4.1 Overview

Understudy extends the GPU's runlist scheduler to open a shadow execution window that lets application B run on the SMs while the UVM driver services application A's page faults (step 1 in Figure 4). When B launches a kernel (2), Understudy reads a per application resource census from the SMs (3), dispatches as many of B's blocks as fit into free SM resources alongside A's stalled blocks (4), and monitors their runtime progress. Whenever no block of B fits initially or freeing extra capacity will shorten B's remaining runtime by more than the cost of saving and restoring state, Understudy saves selected stalled blocks from A and their unaccepted memory requests to the preemption buffer in device memory (5) while the driver continues servicing their faults. Once the driver issues a replay (6), A's resident blocks immediately retry their accesses while A's saved blocks return from the preemption buffer as B's running blocks finish (7), keeping B's shadow ready if A stalls again.

4.2 Admission and Timing

When B submits a kernel during A's fault stall, Understudy first counts how many of B's blocks can fit on the GPU without saving any of A's state. For each SM, the scheduler compares the block requirements in B's launch descriptor against the resource census of registers, shared memory, warp slots, and block slots currently held on that SM, and sums the fitting blocks across all SMs (admitting B

on SMs 1 and 3 in step 1 of Figure 5). If at least one block fits and Understudy has no prior timing history for the kernel, it dispatches the fitting blocks immediately and measures their progress; if prior timings exist, it checks right away whether saving some of A's stalled blocks would finish the kernel faster. When none of B's blocks fit, Understudy proceeds directly to saving stalled blocks from A (Section 4.3), considering any block of A whose warps have all been fault stalled or idle for at least 20 μs while A has unresolved faults. If Understudy has no prior completion rate for B's kernel, it saves all eligible stalled blocks of A unless A is expected to resume before the save completes.

While B runs, Understudy samples its progress every 20 μs by counting how many blocks have finished, remain on the SMs, or wait in the launch queue, and uses the completed blocks to estimate B's block completion rate [55] at the current allocation (falling back to prior timings for the same kernel or the elapsed runtime of the active blocks when no block has finished yet). Understudy caches these measured rates for future launches of the same kernel, indexed by how many blocks ran concurrently, and trusts a rate once two sampling intervals have a standard deviation below 30% of their mean. Whenever B launches a different kernel, a save or restore changes B's allocation, or a fault replay resumes A, the scheduler starts a fresh sampling sequence while keeping earlier records for future launches.

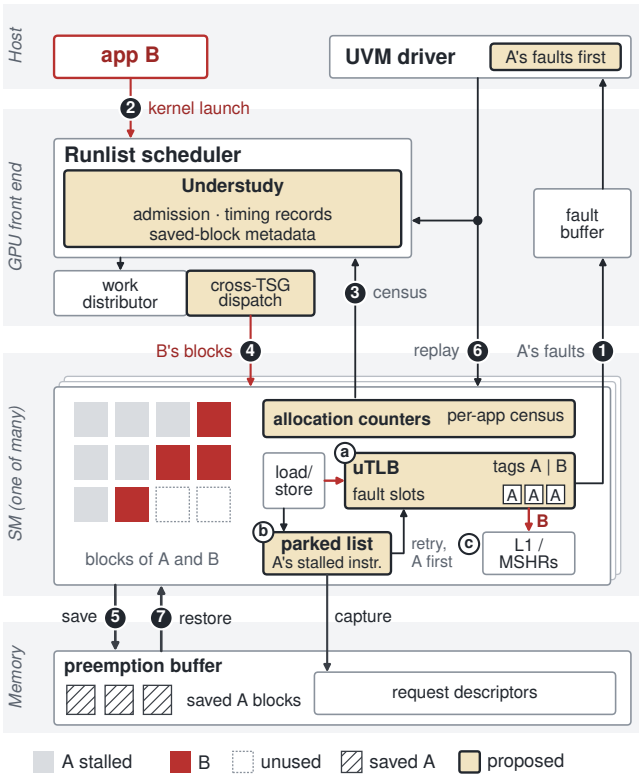


Figure 4: Understudy architecture and event flow during A's fault service.

Table 1: Handling of each operation class when a stalled block is saved and restored.

Operation class	Save and restore
ALU, shuffle, vote, predicate	Issued instructions finish; results stay in registers
Branches, convergence	Issued branches finish; masks and convergence state saved
Loads (all spaces)	Accepted: write registers before save. Unaccepted: captured, reissued first on restore
Stores, atomics, reductions	Accepted: finish before save. Unaccepted: captured with data, reissued once
Block barriers	Arrival state saved; waiting warps resume waiting
Memory fences	Wait for older accepted accesses, or captured in order
Asynchronous copies	Block stays on its SM until replay

4.3 Acquiring Additional Capacity

To decide how many additional blocks of B to make room for, Understudy first counts how many of B's waiting blocks would fit if it saved all eligible stalled blocks of A, and evaluates candidate expansions of one quarter, one half, three quarters, and the full amount against continuing at B's current allocation without saving state. While the full expansion saves all eligible stalled blocks of A, each smaller candidate saves only enough of A's blocks to admit that candidate number of B's blocks, greedily selecting the stalled blocks that transfer the fewest bytes of registers, shared memory, and control state per additional block of B admitted (step 2 of Figure 5).

For each candidate allocation, Understudy predicts B's remaining runtime by dividing its remaining blocks by a verified completion rate recorded at that allocation (or by scaling B's current completion rate in proportion to the candidate block count, or counting remaining concurrent rounds when block duration is known). The scheduler adds the save delay and, when A is expected to resume before B finishes, the restore delay, expanding B's allocation only when the best candidate shortens B's predicted remaining runtime by more than 10%. As B executes, the scheduler repeats this check at each sample while allowing at most two major expansions per kernel of B during A's fault stall (ignoring small saves that free only a few blocks), waiting for any active save or restore to finish before starting a new save, and stopping expansions once the driver finishes servicing A's faults.

To predict when A will resume, the scheduler subtracts the time elapsed in the current stall from a weighted running average of A's previous fault service durations, treating the estimate as reliable once it has observed at least two episodes whose fault counts lie within 30% of the current fault count. When a reliable estimate shows that A will resume in less time than it takes to save and restore A's blocks, Understudy leaves A's blocks on the SMs, and when the estimate is still uncertain but suggests that A is close to resuming, Understudy postpones the save and reconsiders only if A stays stalled past the predicted replay time.

4.4 Saving and Returning Blocks

When a block of A stalls on a page fault, it holds its SM resources and stays on the SM to resume at replay unless Understudy selects it for saving. To save a selected block, Understudy halts instruction issue from its warps and lets already issued non memory instructions and accepted memory requests finish, where a request counts

Table 2: Storage for Understudy's hardware extensions on the modeled RTX PRO 2000 (34 SMs, 48 warps/SM, 4 apps/SM).

Extension	Calculation	Size
Resource census	34 SMs × (4 apps × 7 B counts + 48 warps × 2 B stall time)	4.1 KiB
Fault isolation	34 SMs × (163 tags × 2 b + 48 warps × 4 b + 1 B) + 4,096 L2 TLB tags × 2 b	3.2 KiB
Kernel timing	128 entries[a] × 20 B	2.5 KiB
Block start times	34 SMs × 24 slots × 8 B	6.4 KiB
Request capture	34 SMs × 48 warps × 4 B offset and count	6.4 KiB
Saved block metadata	204 to 816 blocks[b] × 24 B	4.8 to 19.1 KiB
Runlist state	64 entries[a] × 64 B	4.0 KiB
Total on chip (new storage)		31 to 46 KiB
Saved state (device memory)	34 SMs × (256 KiB regs + 100 KiB shared + 21 KiB state) + 864 B/descriptor[c]	≤ 12.5 MiB/app + ≤ 343 MiB pool

[a] Sized above our runs (at most 61 timing entries, four applications). [b] From our largest save (204 blocks) to all 34 × 24 = 816 slots of one application. [c] Section 4.4; a warp needing more than 255 descriptors stays on its SM (our runs peak at 129). Excludes arbitration and dispatch logic.

as accepted once it leaves the SM's translation and dispatch queues for the caches, so accepted loads and atomics write their destination registers before the register file is copied. Because the block's remaining requests are still waiting in the parked list (Figure 4b) or the GPU MMU and have not taken effect, Understudy drains them and captures each one into an 864 B request descriptor from the address, mask, data, and destination register fields in the load/store unit, recording only the unaccepted threads when a warp instruction spans multiple pages or cache lines. Prior GPU preemption mechanisms either wait for fault service before switching contexts, track potentially faulting instructions during execution so they can be replayed [71], drain a block's pending memory requests before saving it (which cannot complete while a fault remains unresolved) [78], or flush only blocks that remain idempotent up to the preemption point [56]. Understudy instead lets another application share the SMs alongside resident fault stalled blocks, adds no instruction tracking during normal execution, and builds request

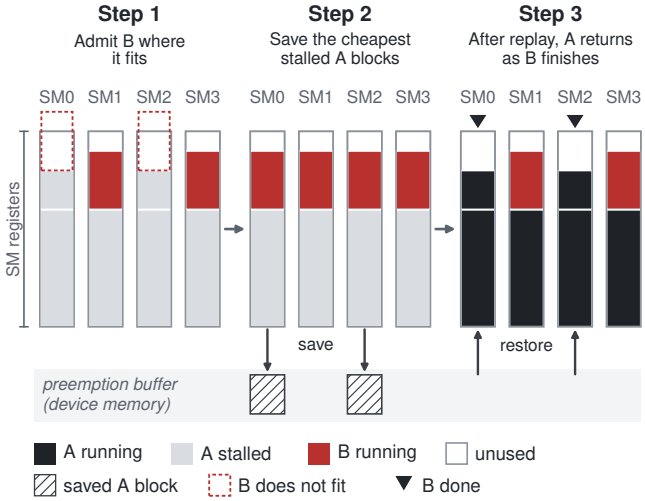


Figure 5: SM register allocation with Understudy.

descriptors only when saving a block (averaging 2.3 descriptors per saved warp, with the hardware cap listed in Table 2). Understudy leaves a block on its SM if it has an in flight asynchronous copy to shared memory or if the preemption buffer is full, and when another stalled block of A keeps a selected block from draining within 5 μs, Understudy drains all of A's stalled blocks on that SM together and keeps any block that still cannot drain on the SM until replay. Once drained, Understudy writes the block's registers, shared memory, warp control state, and descriptors to the preemption buffer to free its SM slots for B.

Because a saved block's fault entries remain in the GPU fault buffer, the driver keeps migrating A's missing pages in the background. At replay, A's resident blocks retry immediately, whereas its saved blocks wait until B has run for at least 500 μs and then reclaim their original SMs as B's blocks finish (step 3 of Figure 5). If 100 μs then pass without any waiting block of A regaining a slot, the scheduler saves the blocks of B holding the needed resources, and we exercise this fallback in Section 6.6. On restore, each warp reloads its state and reissues its captured descriptors in order before fetching its next instruction, and a warp that reached its exit with a captured request retires only after reissuing it so every operation takes effect once (Table 1). We verify this property per lane in simulation (Section 5) and in a TLA+ model [35, 87] checked exhaustively over bounded instances of up to 181,350 states. This is important for cases such as when a replay arrives just as Understudy selects a block whose atomic has reached the cache for lanes 0 to 15 while lanes 16 to 31 wait at translation. There, either the cache accepts the retried lanes and the save waits for their results, or the save moves them into a descriptor first so the replay finds nothing to send and the warp reissues them once on restore, which both our TLA+ model and our directed tests verify.

Finally, if B also faults on the same SM as A and A's unresolved faults fill every micro TLB (uTLB) fault slot so B's warps cannot issue memory requests for 20 μs, Understudy saves B's blocks and defers readmitting B alongside A until the next context switch. When more than two applications share the GPU, a configurable admission depth sets how many neighbors can run alongside A at once, with each admitted neighbor holding its slot until its last block leaves the SMs (Section 6.6 evaluates depths one and two, while our single neighbor experiments run at depth one). At depth one, if B stalls on faults and a third application C arrives with non-faulting work, Understudy saves B's stalled blocks before admitting C, whereas at higher depths it admits C directly alongside A and B and saves the longest stalled neighbor only when all admission slots are full or C needs more SM resources.

4.5 Hardware Support and Overhead

To let B execute while A's blocks remain on the SMs, Understudy extends the hardware work distributor to dispatch blocks across TSGs while using the subcontext support in Section 2.2 to keep each application in its own address space. Along each SM's memory pipeline, Understudy tags uTLB entries, walk slots, and fault slots with a 2 bit application ID (Figure 4a) so that a fault from A stalls translation only for A while B's hits and page walks continue. The load/store unit moves any instruction stalled on a held fault out of its dispatch register into a parked list (Figure 4b) so B's accesses

Table 3: Evaluation setup (m: transferred state in MiB).

Parameter	Value
Hardware	
GPUs	RTX PRO 4000 SFF (70 SMs, 24 GB); RTX PRO 2000 (34 SMs, 16 GB) for calibration and NVBit [75] 1.8 traces
Simulated GPU (RTX PRO 2000 at 2.810 GHz)	
Resources per SM	1,536 threads, 48 warp slots, 24 block slots, 65,536 registers, 128 KiB unified L1 and shared memory (up to 100 KiB shared), 3 uTLB fault slots, 32 walk slots
Memory	32 MB L2, 288 GB/s DRAM, 4,096 entry L2 TLB (100 cycle walk); DRAM load 660 cycles (646 to 665 on hardware), L2 hit 288 cycles (314 on hardware)
Fault service	
Driver batch processing	2 μs + 13 μs per entry, up to 256 entries per batch
Fault signaling	20 μs to reach the driver, 10 μs batch fetch, 10 μs replay
Page migration	64 KiB chunks, 25 GB/s, 2 μs latency, tree based prefetch
Scheduler	
Slice and launch	2,031 μs quantum, 15 μs arrival latency, 5 μs launch gap
Save state	$62.4 + 2.75\,m$ μs
Restore, neighbor done	$56.8 + 2.13\,m$ μs
Restore, immediate	$12.6 + 2.6\,m$ μs

Transfer models fit on the RTX PRO 2000 over 18 configurations (varying registers, warps, shared memory, and SMs; 1.1 to 11.8 MiB of state): RMS error 5.5 μs for saving and 1.9 μs for restoring after a neighbor finishes; within 5 μs of the 81 to 134 μs hardware time slice switch.

can issue past it, while a retry arbiter (Figure 4c) reissues parked instructions after replay with priority for A and the UVM driver services A's faults before those of admitted neighbors. Because requests enter the caches only after translation, A's faulted accesses occupy no L1 MSHRs, interconnect buffers, or L2 entries. B still shares the caches, the L2 TLB, and the pool of fault slots with A, and if A's faults hold every fault slot on an SM while B also faults, the fallback in Section 4.4 saves B's blocks. Finally, Understudy extends the SM allocation counters with per application counts of resident and fault stalled warps so the scheduler can locate free capacity and select stalled blocks to save. Altogether, Understudy's on chip structures require 31 to 46 KiB of storage (Table 2, or about 0.5% of the 8.5 MiB register files), while saved blocks reuse the driver's existing context switch preemption buffer in device memory (up to 12.5 MiB of block state per application plus a descriptor pool of at most 343 MiB, together under 2.3% of the 16 GB device memory).

5 Experimental Methodology

We evaluate Understudy by combining hardware experiments on NVIDIA RTX PRO 2000 and RTX PRO 4000 Blackwell GPUs with a cycle level simulator that we built on Accel-Sim [32] and calibrated against the RTX PRO 2000 (Table 3) and validated against it (Table 5). In our hardware runs, we correlate B's kernel launch and GPU cycle counter timestamps with fault logs from an instrumented UVM driver. In simulation, we run each application in its own context across shared SMs and caches, and our demand paging model stalls warps that fault on CPU resident pages until the simulated UVM driver migrates their pages over PCIe and issues a replay. Unless noted, we start each run with A's managed pages in CPU memory so A faults them in on first access without page eviction. To verify that saving and restoring blocks never drops or repeats an operation, we check retired instruction counts on every run and maintain a per lane ledger in the simulator to confirm that every memory effect of a saved block happens once across capture, replay, and restoration. Because the ledger and instruction counts check that operations execute once rather than checking the data values they produce, we also run 32 directed functional and mutation tests covering all operation classes in Table 1. These tests functionally execute loads, stores, atomics, fences, and barriers across a save and restore cycle to check the data values returned by memory and the order in which captured requests reissue. We also verify that the suite catches injected simulator bugs such as dropping a warp lane, repeating an atomic update, or reordering reissued requests.

Table 5 compares our simulator against RTX PRO 2000 measurements on the three quantities that drive our results. We run each faulting workload in its traced configuration to record its fault episode from the unified memory trace and its stalled SM occupancy from 10 kHz GPU telemetry, and we time each arriving workload alone. Simulated arriving kernels finish within 9% of hardware, stalled occupancy matches within a few points outside of gather b256 (about 10 points higher in simulation), and the gather and DLRM episodes run 17 to 26% longer than measured. MoE is the one exception on episode length because the hardware prefetcher streams its sequential expert weights in 2 MB blocks after the initial faults, whereas our model services each 64 KB chunk separately and stretches the 4.0 ms episode fourfold, inflating default scheduling's delay on the MoE pairs. Finally, whereas hardware starts Qwen's 41 queued kernels about 0.3 μs apart, the simulator adds a fixed 5 μs launch gap before each kernel (stretching the full layer from 622 to 859 μs), and we evaluate the sensitivity of our results to both simulator settings in Section 6.1.

Table 4: Evaluated workloads (a) and pairs (b). Held: share of SM warp slots and registers that A occupies when B arrives. Panel (b): blocks of B's first large kernel that fit beside A, out of those B runs on an idle GPU.

(a) Applications

	Application	Description	Memory (GiB)*	Launches	Alone	Held
A	DLRM [45]	Recommendation inference, batch 256	8.7 / 5.96 / 0.56	25 (13)	173.1 ms	79 / 89%
A	MoE [64]	Eight expert, top two layer step, 96 MB	0.26 / 0.03 / 0.02	6 (2)	16.0 ms	25 / 96%
A	Gather b256	PyTorch [57] EmbeddingBag lookup, batch 256	4.3 / 4.0 / 0.11	2 (2)	26.8 ms	66 / 74%
A	Gather b1024	First kernel of a four table lookup	12.1 / 3.0 / 1.48	1 (1)	255.8 ms	63 / 59%
B	Qwen [85]	Qwen2.5-3B decoder layer, batch 1	6.3	41 (18)	0.86 ms	
B	CAGRA [54]	cuVS approximate nearest neighbor search, 64 queries	1.3	2 (2)	0.24 ms	
B	LULESH [31]	Hydrodynamics stress integration, size 16	0.2	1 (1)	0.19 ms	

(b) B's blocks that fit beside A

A \ B	Qwen	CAGRA	LULESH
DLRM	19/204	1/64	
MoE	5/204	0/64	
Gather b256			0/68
Gather b1024		42/64	34/68

*For each faulting application A, the memory column lists peak resident GPU memory (live), the managed allocation touched by the traced kernels (traced), and the data volume migrated during the run (migrated), while the launches column lists total launches followed by distinct kernels in parentheses. Every evaluated pair fits within the 15.9 GiB of usable device memory on the RTX PRO 2000 and peaks at 15.0 GiB when pairing DLRM with Qwen.

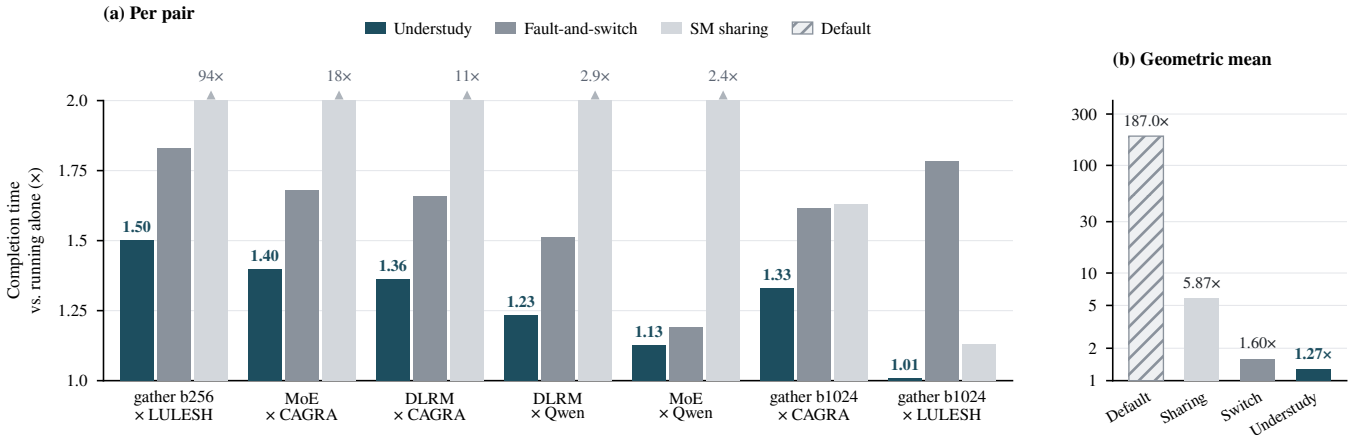


Figure 6: Completion time of work submitted during a neighbor's fault service, relative to running alone.

We evaluate the seven workload pairs in Table 4 alongside the multi launch, continuous service, and multi application scenarios in Section 6. We compare Understudy against default scheduling, which waits for SM fault service to finish before switching contexts, as well as SM sharing, which runs B only in the resources A leaves free, and fault and switch, which saves all of A's resident blocks before starting B (using the same 500 μs return window from Section 4.4). In our primary configuration, all policies replay identical traces and submit B at the same point early in A's fault stall (0.5 to 52 ms after A starts, or at most 20% into its episode), and we track B's completion time relative to running alone alongside A's total completion time relative to default scheduling.

6 Evaluation

6.1 Application Performance

We first examine how Understudy performs when an application submits work during a neighbor's page fault service (Figure 6). Across our seven workload pairs, the arriving application's geometric mean completion time relative to running alone drops from 187× under default scheduling and 5.87× under SM sharing to 1.60× under fault and switch and 1.27× under Understudy (a 147× speedup over default scheduling). Understudy finishes the arriving work faster than fault and switch in every pair (1.26× faster in geometric mean) while transferring 44% less total execution state (78.0 versus 139.1 MiB). In pairs such as gather b1024 with LULESH, where B fits into free resources alongside A, Understudy saves no state and finishes LULESH within 1% of its isolated runtime, whereas in pairs such as DLRM with CAGRA, where SM sharing slows CAGRA to 11.2×, Understudy saves only enough stalled DLRM blocks to finish CAGRA at 1.36× without preempting DLRM's entire context.

Across all seven pairs, A finishes in 0.98 to 1.02× its default scheduling runtime, adding at most 1.6% overhead (in MoE with CAGRA) in this configuration. When we vary B's arrival across 10, 50, and 90% of A's fault episode, Understudy reaches geometric mean completion times of 1.32, 1.21, and 1.22×, beating both fault and switch (1.65, 1.64, and 1.61×) and fixed allocation (Section 6.3; 1.38, 1.28, and 1.29×).

We also test the two simulator parameters that differ from hardware in Section 5. Shortening Qwen's inter kernel launch gap from 5 to 0.3 μs brings its standalone layer time to 664 μs (versus 622 μs on hardware) and keeps Understudy fastest for Qwen, finishing at 1.17× alongside DLRM (versus 1.23× under SM sharing and 1.71× under fault and switch) and at 1.17× alongside MoE (versus 1.26× under fault and switch). When we instead lower the per fault service cost so that MoE's fault episode matches hardware (4.3 versus 4.0 ms), Understudy still finishes both Qwen and CAGRA fastest (1.38× and 1.37× versus 1.53× and 1.65× under fault and switch). In that setting, MoE's episode consists of 0.26 ms fault batches between which its blocks resume to issue new faults. While CAGRA finishes within about one batch without delaying MoE (0.98×), Qwen's layer, which takes 1.2 ms alongside MoE, spans several batches before MoE's saved blocks can resume, causing MoE to finish 17% later under Understudy than under default scheduling (compared to 22% later under fault and switch).

Table 5: Comparison of our simulator against RTX PRO 2000 hardware, where F marks measurements used to set a model constant and H marks held out checks. *Arriving kernel alone* excludes launch overhead, while *held while stalled* reports the share of SM warp slots, registers, and block slots occupied during the first touch kernel's stall (or expert GEMV kernels for MoE).

Quantity	Workload	Hardware	Simulated	Sim/HW	Fit
Arriving kernel alone	LULESH k2	185 μs	184 μs	1.00	F
	CAGRA search	213 μs	232 μs	1.09	H
	Qwen decoder layer	613 μs	652 μs	1.06	F
Held while stalled	gather b1024	61, 57, 15%	59, 55, 15%		H
	gather b256	56, 63, 14%	66, 74, 17%		H
	DLRM b256	53, 59, 13%	56, 63, 15%		H
	MoE 96 MB GEMV	23, 92, 23%	23, 90, 23%		H
First touch episode	gather b256	22.5 ms	26.8 ms	1.19	H
	gather b1024	219 ms	256 ms	1.17	F
	DLRM b256	138 ms	173 ms	1.26	H

6.2 Capacity Acquisition and State Saving

When B needs more capacity than A leaves free, Understudy saves only the stalled blocks required rather than A's entire context (Figure 7), moving a median of 9.1 MiB of combined save and restore state (registers, shared memory, and control state) across the seven pairs compared with 16.8 MiB under fault and switch. For CAGRA alongside DLRM, Understudy saves and restores 4.56 MiB in each direction versus 8.43 MiB under fault and switch (46% less), and when Qwen's varying kernel footprints trigger two saves alongside DLRM, Understudy moves 28.5 MiB in total versus 50.5 MiB (43% less) while finishing Qwen at 1.23× rather than 1.51× its isolated runtime. Only in MoE with Qwen, where MoE holds 96% of the registers, does Understudy move slightly more state than fault and switch (17.2 versus 16.8 MiB), while still finishing Qwen faster at 1.13× versus 1.19×.

To verify that memory contention from state transfers does not alter these results, we rerun all seven pairs with every save and restore, including its request descriptors, issued as copy engine requests through the simulated L2 and DRAM at the measured 2.75 and 2.13 μs/MiB rates (charging the setup latency once per acquisition episode). Memory contention raises B's geometric mean completion time from 1.27× to 1.32× under Understudy, from 1.28× to 1.34× under whole SM saving (Section 6.3), from 1.36× to 1.40× under whole context switching, and from 1.60× to 1.63× under fault and switch, leaving Understudy 1.23× faster than fault and switch while keeping A's runtime within 1.6% of default scheduling. Bypassing L2 so state transfers go directly to DRAM yields geometric means of 1.31× for Understudy, 1.37× for whole SM saving, and 1.65× for fault and switch, and Understudy likewise stays ahead when we halve or double copy engine bandwidth. Only if every incremental save within an acquisition episode paid a fresh 62.4 μs setup cost would MoE with Qwen slow from 1.19× to 1.34× (versus 1.24× for fault and switch), with the other six pairs unchanged.

6.3 Sources of Improvement

To separate the benefit of Understudy's admission and timing policy from its per block saving granularity, we first vary how much of A's state each save moves. Our **whole context switching** ablation uses Understudy's share first admission and online timing policy to decide when to save, but moves A's entire context like fault and switch. Sharing free resources first and saving only when beneficial already lowers B's geometric mean completion time from 1.60× under fault and switch to 1.36×, and per block saving cuts it a further 6.6% to 1.27× (dropping gather b256 with LULESH from 1.83× to 1.50×). An intermediate **whole SM saving** ablation clears all of A's blocks from each chosen SM rather than only the blocks B needs, nearly matching Understudy in completion time (1.28 versus 1.27×) while moving 36% more state (106.3 versus 78.0 MiB) and adding memory contention (Section 6.2) by saving blocks whose resources B never uses. These results show that while fault isolation and block suspension enable execution during fault service, our admission and timing policy drives most of the speedup and per block selection primarily reduces state transfer traffic.

We next keep per block saving and replace our online timing policy with static rules. A **fixed allocation** ablation saves A's blocks whenever B's kernel does not fully fit without weighing the save cost, much like prior fault triggered block switching [33]. It matches Understudy in six pairs, but in gather b1024 with LULESH it needlessly moves 7.8 MiB though LULESH fits in free resources, slowing LULESH from 1.01× to 1.46×. A static **threshold policy** shares free resources first and saves stalled blocks only when the fitting fraction of B's blocks falls below a threshold. After tuning on three development pairs at the 50% arrival point (selecting 75% of 25 to 100%), this policy reaches a geometric mean of 1.37× (1.32× at 100%) across the 18 unseen pair and arrival combinations, compared to 1.24× for Understudy and 1.63× for fault and switch. Because a static threshold ignores remaining fault service time, when B arrives at 90% of the episode it saves state right before A resumes in gather b256 with LULESH (2.59 versus 1.95×) and leaves B sharing limited resources in MoE with CAGRA (1.74 versus 1.19×).

6.4 Adaptation Across Launches

We next submit kernel sequences with changing resource requirements, spacing consecutive launches by 1.5 times the previous launch's isolated runtime. When three CAGRA searches alternate with two LULESH kernels alongside MoE, Understudy cuts the summed latency of the five launches from 66.04 ms under default scheduling and 21.25 ms under SM sharing to 1.45 ms, beating fault and switch (1.84 ms) by 21% as it adjusts its allocation between the two kernels. Alongside gather b1024, Understudy likewise reduces the five launch total from 1,019 ms under default scheduling to 1.36 ms, finishing 1.15× faster than SM sharing (1.57 ms) and 1.27× faster than fault and switch (1.72 ms).

We also run four consecutive launches of the same kernel to show how Understudy adapts to A's register footprint. Alongside gather b1024, enough registers remain free on every SM to fit a LULESH block, so Understudy and SM sharing finish the four LULESH launches in 0.75 and 0.77 ms versus 815 ms under default scheduling and 1.20 ms under fault and switch. Alongside gather b256, whose higher register use per thread leaves almost no room for LULESH, SM sharing takes 73.26 ms while Understudy saves stalled blocks twice and finishes in 1.18 ms, beating both default scheduling (96.26 ms) and fault and switch (1.29 ms). Across four CAGRA searches, Understudy finishes the first search about 70 μs faster than fault and switch in both pairs, completing all four 1.31×

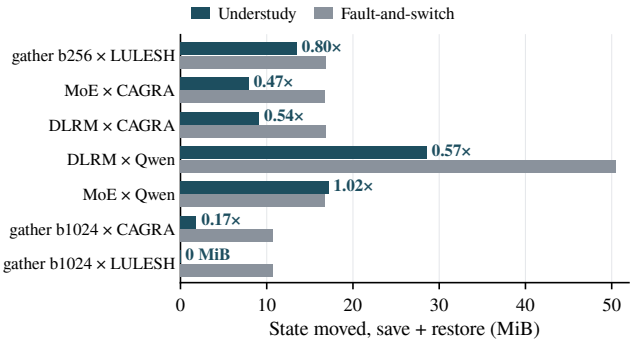


Figure 7: Execution state saved and restored per pair; labels give Understudy's state relative to fault and switch.

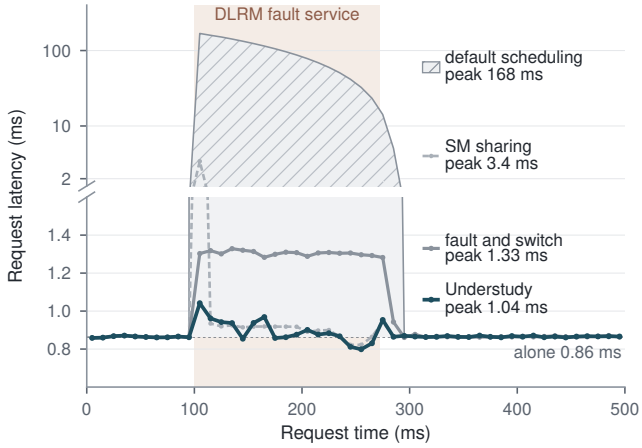


Figure 8: Qwen decoder layer request latency during DLRM's fault service.

faster alongside gather b1024 (1.10 versus 1.44 ms) and within 1% alongside MoE (1.28 versus 1.27 ms).

6.5 Continuous Service

To connect our single launch results to the continuous serving scenario of Figure 1, we issue 50 Qwen decoder layer requests every 10 ms from 5 to 495 ms while DLRM's first forward pass arrives at 100 ms and services 12,261 page faults over 172 ms under default scheduling (Figure 8). Because simulating all 1,543 kernels of a full token is infeasible at the cycle level, each request runs one of Qwen's 36 identical decoder layers (41 dependent launches taking 0.86 ms alone), preserving a decode step's kernel mix and launch dependencies at a lighter GPU load than running all 36 layers back to back. Under default scheduling, the request at 105 ms waits for the entire fault episode and delays 19 of the 50 requests past twice their isolated latency (up to 168 ms). Fault and switch completes requests during DLRM's fault service in 1.28 to 1.33 ms (1.5× alone) while preempting DLRM's entire context 54 times, and SM sharing spikes to 3.4 ms on the request that coincides with DLRM's launch. Understudy serves every request within 1.04 ms (1.2× alone, accumulating only 0.9 ms of total excess latency versus 0.8 ms under whole SM saving) while finishing DLRM's pass in 172 ms compared to 176 ms under default scheduling and 182 ms under fault and switch. Even when 50% or 90% of DLRM's pages start resident, default scheduling still stalls requests for up to 83 and 14 ms, whereas Understudy keeps every request within 1.11 ms. To test higher loads closer to a full decoder running layers back to back, we shorten the gap between requests to 2 ms (43% GPU load) and 1 ms (86% load), issuing requests from 50 to 350 ms. At 2 ms intervals, Understudy finishes all 87 requests issued during DLRM's fault episode within 1.34 ms, compared to 1.51 ms under fault and switch and 173 ms under default scheduling. At 1 ms intervals, where each 0.86 ms request leaves little headroom across the first 108 ms of DLRM's episode (109 requests), Understudy builds a brief queue while first saving DLRM's stalled blocks, peaking at 4.5 ms before draining the queue within 30 ms. By contrast, fault and switch pays a full context switch on every request and falls behind the 1 ms arrival rate, growing its queue to 33 ms.

6.6 Multiple Faulting Applications

We next evaluate what happens when more than one application triggers page faults while sharing the GPU. We run two faulting applications, gather b256 and hash probe, together under demand paging (with a 12.6 GiB combined footprint that fits in device memory) while a third application submits ten LULESH launches at 2 ms intervals. Although a LULESH launch takes 190 μs alone, its first launch waits 880 ms under default scheduling until both faulting applications finish. Fault and switch keeps every LULESH launch at 330 μs (1.74× alone) but performs 524 whole context switches as the two faulting applications repeatedly preempt each other on every stall, slowing gather b256 from 26.8 to 266.9 ms (and still performing over 500 switches even if switching back waits for a full time slice). By contrast, Understudy serves LULESH at a median of 208 μs (1.10× alone and at most 294 μs) with just six block saves moving 19.4 MiB, while both faulting applications finish within 1% of their default scheduling runtimes.

To stress the admission depth (neighbors admitted alongside A; Section 4.4) and the block return path with two competing neighbors, we submit 24 LULESH launches every 300 μs alongside 17 CAGRA searches so that two LULESH blocks and one CAGRA block occupy each SM when gather b256's faults finish servicing. Default scheduling delays the first LULESH and CAGRA launches by 879 and 872 ms, SM sharing delays one CAGRA search by 864 ms, and fault and switch performs 445 whole context switches that slow gather b256 to 247 ms. At admission depths one and two, Understudy finishes both faulting applications as fast as default scheduling while serving LULESH at a median of 188 and 208 μs and CAGRA at 230 and 267 μs. Raising the depth to two lets LULESH and CAGRA run concurrently, cutting the longest CAGRA search from 1.6 to 0.86 ms. Because active neighbor blocks finish within a few hundred microseconds, A's saved blocks return without triggering the 100 μs timeout. To bound A's cost when a forced return does fire, we pair gather b256 with a feature hashing kernel whose blocks are still running when A's saved blocks become ready. Once the timeout expires, Understudy saves all 128 of B's blocks (1,024 warps) so A can return, finishing B in 4.0 ms instead of 26.8 ms while A takes 4.8% longer than under default scheduling (4.0% under fault and switch). Across this forced return, we verify that both applications execute the exact same instructions as under default scheduling.

7 Discussion

While Understudy prevents application A's page fault stalls from blocking a colocated application B whose data is already in GPU memory, A itself still pays the PCIe latency of migrating its missing pages from the CPU, so operators still need to give A enough GPU memory to keep its own migration cost acceptable. Moreover, A rarely has independent work of its own that can fill those stalls. Kernels that follow a faulting kernel in the same stream usually depend on its output (as in DLRM), while launching independent kernels from A's other streams typically touches new pages and triggers further faults [33]. When A does have ready kernels whose

data is already resident, Understudy can admit them using the same block saving mechanism that it uses for a colocated application.

Second, in our simulated runs, the combined memory footprint of A and B fits within the 16 GB GPU, so A faults only when it first touches its managed pages. When A's hot working set instead exceeds its available GPU memory, the driver must evict resident pages to bring in new ones, causing A to fault repeatedly throughout its run. On our RTX PRO 2000, for example, capping a 12 GB DLRM model at a 2.3 GB GPU memory budget beside the Qwen decoder keeps DLRM's 2.6 to 3.0 GB hot working set from fitting in memory. The resulting eviction and fault episodes raise the decoder's 99th percentile step latency from 26 ms alone to between 191 and 211 ms, whereas avoiding those faults with remote mapping or per batch prefetching cuts DLRM's throughput by 20 to 34%. Although our simulator does not model page eviction, each recurring fault stall presents the hardware scheduler with the same resource trade off as a first touch stall, allowing Understudy to apply the same admission and block saving mechanism to every episode.

Finally, this memory limit is also why we do not pair our BFS traversal (13.1 GiB live for a 12.0 GiB graph) with the 6.3 GiB Qwen decoder, since together they exceed the 15.9 GiB of usable memory on the RTX PRO 2000. When the graph does not fit in free GPU memory, demand paging thrashes on our RTX PRO 2000 and migrates 101 GB of data, about eight times the graph size, across a 20 s traversal, whereas mapping the graph remotely in CPU memory [44] finishes 10 to 16× faster without triggering page faults. When we instead run a smaller 7.5 GB graph that fits beside the decoder, BFS faults its pages in once during a 0.76 s first traversal, behaving just like our seven evaluated pairs.

8 Related Work

GPU exception handling and preemption. To make faulting GPU code restartable so an SM can switch contexts during page migration, iGPU [43] re executes compiler constructed idempotent regions, whereas Tanasic et al. [71] handle exceptions in hardware by stalling a warp after each potentially faulting instruction, buffering those instructions, or logging their operands. For general kernel preemption [37, 70], FLEP [79] combines compiler transformations with runtime scheduling to yield selected SMs, while Chimera [56] decides from estimated cost whether to save, drain, or flush each resident block, flushing only blocks that remain idempotent up to the preemption point. To speed up a faulting kernel's own paging, Kim et al. [33] oversubscribe thread blocks so that when every warp of a block stalls on page faults, the SM swaps in another block from the same kernel to expose more faults for driver batching. Similarly, an Intel design [34] preempts a multi cluster compute block on a page fault and loads another context's work onto it while average fault latency remains high. Whereas these approaches either vacate whole SMs or compute blocks or swap in blocks from the faulting kernel itself, Understudy runs an independent application alongside resident fault stalled blocks and saves individual stalled blocks at fault time without runtime instruction tracking or compiler idempotence requirements.

Concurrent execution and resource partitioning. SMK [78] uses block level preemption and warp scheduling to share SMs across active kernels, Warped-Slicer [83] partitions SM resources using online profiling and an analytical model, and Zorua [74] virtualizes registers, shared memory, and thread slots. Within an application, replayable far faults [90] and selective fault stalling [24] let warps that are not waiting on a fault continue issuing instructions. Unlike spatial sharing among kernels that can all freely access memory, Understudy targets page fault stalls where parked blocks hold registers and slots while their unresolved requests block ordinary preemption. Software schedulers instead preempt GPU work by shortening TSG time slices [22], throttling or interrupting submitted commands [66], resetting idempotent kernels [25], or splitting kernels across launches [27], though any scheme that relies on ordinary context switching inherits the unresolved SM fault limitation from Section 2.3.

Colocation and GPU memory management. Prior unified memory studies characterize demand paging overheads [4, 5] and shorten an application's own fault stalls by predicting or placing pages ahead of use [8, 16, 30, 65, 90] and by guiding eviction and thread throttling under memory pressure [10, 23, 36]. Deep learning runtimes similarly hide transfer latency by swapping or offloading tensors explicitly across the host interconnect [28, 29, 58, 62, 63, 67]. Because those paging techniques shorten fault episodes rather than eliminating them, Understudy complements them by letting colocated work use the GPU during the stalls that remain. Across concurrent applications, MASK, Mosaic, and page walk stealing [6, 7, 9] reduce address translation overheads and interference. At the runtime level, PILOT [61] curbs UVM thrashing by delaying or preempting kernels or pinning excess allocations in host memory, SIRIUS [76] reclaims background training memory as foreground inference demand fluctuates, and software sharing systems [3, 41, 80, 84] multiplex GPU compute and memory across jobs. Rather than throttling a faulting job or shrinking its memory, Understudy removes the hardware scheduling bottleneck during fault service so colocated work already resident in GPU memory can run alongside or in place of fault stalled blocks.

What Understudy brings to the table. While prior work has explored fault triggered block switching, selective preemption on faults, and continued execution during faults individually, Understudy brings them together across independent applications. It lets B run while A's faults remain unresolved, captures pending requests at save time so stalled blocks resume without runtime logging or instruction re execution, and saves only the stalled blocks whose removal pays off within the remaining fault service time (which we compare against fixed allocation in Section 6.3).

9 Conclusion

On the two Blackwell GPUs we measured, unresolved page faults block not only the application waiting on missing pages, but also colocated applications whose working sets already reside in GPU memory. Understudy reclaims these fault stall intervals for colocated work by placing arriving blocks into unused SM resources alongside fault stalled blocks, using online progress measurements to save selected stalled blocks only when the expected reduction in execution time outweighs the state transfer cost, and coordinating block restoration as the driver finishes fault service. In a cycle level simulator calibrated against hardware measurements on an RTX PRO 2000, Understudy reduces the geometric mean completion time

of work submitted during a neighbor's fault service from 187× to 1.27× isolated execution across seven workload pairs while adding at most 1.6% to the faulting application's completion time over default scheduling in our primary configuration. While our simulator models first touch fault stalls, we expect the same mechanism to carry over to recurring faults under sustained oversubscription and defer a full characterization of that regime to future work.

Acknowledgments. We used generative AI tools (OpenAI ChatGPT and Codex, and Anthropic Claude) to assist with drafting and revising text, code, experimental scripts, tables, and figures. We reviewed all generated material, verified all sources and results, and take full responsibility for the work.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[2] Jacob T. Adriaens, Katherine Compton, Nam Sung Kim, and Michael J. Schulte. 2012. The case for GPGPU spatial multitasking. In *IEEE International Symposium on High-Performance Comp Architecture*. 1–12. doi:10.1109/HPCA.2012.6168946

[3] Georgios Alexopoulos and Dimitris Mitropoulos. 2024. nvshare: Practical GPU Sharing without Memory Size Constraints. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings* (Lisbon, Portugal) *(ICSE-Companion '24)*. Association for Computing Machinery, New York, NY, USA, 16–20. doi:10.1145/3639478.3640034

[4] Tyler Allen, Bennett Cooper, and Rong Ge. 2024. Fine-grain Quantitative Analysis of Demand Paging in Unified Virtual Memory. *ACM Trans. Archit. Code Optim.* 21, 1, Article 14 (Jan. 2024), 24 pages. doi:10.1145/3632953

[5] Tyler Allen and Rong Ge. 2021. In-depth analyses of unified virtual memory system for GPU accelerated computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 64, 15 pages. doi:10.1145/3458817.3480855

[6] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: a GPU memory manager with application-transparent support for multiple page sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 136–150. doi:10.1145/3123939.3123975

[7] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J. Rossbach, and Onur Mutlu. 2018. MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 503–518. doi:10.1145/3173162.3173169

[8] Pratheek B, Guilherme Cox, Jan Vesely, and Arkaprava Basu. 2024. SUV: Static Analysis Guided Unified Virtual Memory. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 293–308. doi:10.1109/MICRO61859.2024.00030

[9] Pratheek B, Neha Jawalkar, and Arkaprava Basu. 2021. Improving GPU Multitenancy with Page Walk Stealing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 626–639. doi:10.1109/HPCA51647.2021.00059

[10] Pratheek B, Khushit Shah, and Arkaprava Basu. 2026. Observability-Aided GPU Memory Oversubscription. In *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*. 1–17. doi:10.1109/ISCA66397.2026.00108

[11] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 499–514. https://www.usenix.org/conference/osdi20/presentation/bai

[12] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 163–174. doi:10.1109/ISPASS.2009.4919648

[13] Joshua Bakita and James H. Anderson. 2023. Hardware Compute Partitioning on NVIDIA GPUs. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 54–66. doi:10.1109/RTAS58335.2023.00012

[14] Joshua Bakita and James H. Anderson. 2024. Demystifying NVIDIA GPU Internals to Enable Reliable GPU Management. In *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 294–305. doi:10.1109/RTAS61025.2024.00031

[15] Joshua Bakita and James H. Anderson. 2025. Hardware Compute Partitioning on NVIDIA GPUs for Composable Systems. In *37th Euromicro Conference on Real-Time Systems (ECRTS 2025) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 335)*, Renato Mancuso (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 21:1–21:25. doi:10.4230/LIPIcs.ECRTS.2025.21

[16] Trinayan Baruah, Yifan Sun, Ali Tolga Dinçer, Saiful A. Mojumder, José L. Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, and David Kaeli. 2020. Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 596–609. doi:10.1109/HPCA47549.2020.00055

[17] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A quantitative study of irregular programs on GPUs. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC) (IISWC '12)*. IEEE Computer Society, USA, 141–151. doi:10.1109/IISWC.2012.6402918

[18] Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. 2018. Deadline-Based Scheduling for GPU with Preemption Support. In *2018 IEEE Real-Time Systems Symposium (RTSS)*. 119–130. doi:10.1109/RTSS.2018.00021

[19] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. doi:10.1109/IISWC.2009.5306797

[20] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 681–696. doi:10.1145/2872362.2872368

[21] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) benchmark suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (Pittsburgh, Pennsylvania, USA) *(GPGPU-3)*. Association for Computing Machinery, New York, NY, USA, 63–74. doi:10.1145/1735688.1735702

[22] Ruwen Fan, Tingxu Ren, Minhui Xie, Shiwei Gao, Jiwu Shu, and Youyou Lu. 2025. GPREEMPT: GPU Preemptive Scheduling Made General and Efficient. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association, Boston, MA, 263–272. https://www.usenix.org/conference/atc25/presentation/fan

[23] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2019. Interplay between hardware prefetcher and page eviction policy in CPU-GPU unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 224–235. doi:10.1145/3307650.3322224

[24] Olivier Giroux and Shirish Gadre. 2017. Selective Fault Stalling for a GPU Memory Pipeline in a Unified Virtual Memory System. US Patent 9,830,224, granted 2017 (NVIDIA Corporation; priority 2013).

[25] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 539–558. https://www.usenix.org/conference/osdi22/presentation/han

[26] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. 2008. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. Association for Computing Machinery, New York, NY, USA, 260–269. doi:10.1145/1454115.1454152

[27] Tiancheng Hu, Chenxi Wang, Ting Cao, Jin Qin, Lei Chen, Xinyu Xiao, Junhao Hu, Hongliang Tian, Shoumeng Yan, Huimin Cui, Quan Chen, and Tao Xie. 2026. Hummingbird: SLO-Oriented GPU Preemption at Microsecond-scale. arXiv:2601.04071 [cs.DC] https://arxiv.org/abs/2601.04071

[28] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. SwapAdvisor: Pushing Deep Learning Beyond the GPU Memory Limit via Smart Swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1341–1355. doi:10.1145/3373376.3378530

[29] Ranggi Hwang, Jianyu Wei, Shijie Cao, Changho Hwang, Xiaohu Tang, Ting Cao, and Mao Yang. 2024. Pre-Gated MoE: An Algorithm-System Co-Design for Fast and Scalable Mixture-of-Expert Inference. In *Proceedings of the 51st Annual International Symposium on Computer Architecture* (Buenos Aires, Argentina) *(ISCA '24)*. IEEE Press, 1018–1031. doi:10.1109/ISCA59077.2024.00078

[30] Jaehoon Jung, Jinpyo Kim, and Jaejin Lee. 2023. DeepUM: Tensor Migration and Prefetching in Unified Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 207–221. doi:10.1145/3575693.3575736

[31] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary Devito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles H. Still. 2013. Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 919–932. doi:10.1109/IPDPS.2013.115

[32] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 473–486. doi:10.1109/ISCA45697.2020.00047

[33] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. 2020. Batch-Aware Unified Memory Management in GPUs for Irregular Workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1357–1370. doi:10.1145/3373376.3378529

[34] Altug Koker, Ingo Wald, David Puffer, Subramaniam M. Maiyuran, Prasoonkumar Surti, Balaji Vembu, Guei-Yuan Lueh, Murali Ramadoss, Abhishek R. Appu, and Joydeep Ray. 2019. Page Faulting and Selective Preemption. US Patent 10,282,812, granted 2019 (Intel Corporation; priority 2017).

[35] Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., USA.

[36] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. 2019. A Framework for Memory Oversubscription Management in Graphics Processing Units. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 49–63. doi:10.1145/3297858.3304044

[37] Zhen Lin, Lars Nyland, and Huiyang Zhou. 2016. Enabling efficient preemption for SIMT architectures with lightweight context switching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '16)*. IEEE Press, Article 77, 11 pages.

[38] Fangyue Liu, Hua Liu, Xinyuan Lyu, Shuo Ai, Hao Liang, Lingpeng Chen, Ziqian Hu, Chong Zha, Xin Jin, Hanmei Luo, and Peng Chen. 2026. Valve: Production Online-Offline Inference Colocation with Jointly-Bounded Preemption Latency and Rate. arXiv:2604.07874 [cs.OS] https://arxiv.org/abs/2604.07874

[39] Rixin Liu, Xingqi Cui, Kaijian Wang, Xinheng Ding, Zirui Liu, Yuke Wang, and Jiarong Xing. 2026. Characterization-Guided GPU Fault Resilience in NVIDIA MPS. arXiv:2605.26461 [cs.DC] https://arxiv.org/abs/2605.26461

[40] Weiguo Liu, Bertil Schmidt, Gerrit Voss, and Wolfgang Müller-Wittig. 2008. Accelerating molecular dynamics simulations using Graphics Processing Units with CUDA. *Computer Physics Communications* 179, 9 (2008), 634–641. doi:10.1016/j.cpc.2008.05.008

[41] Yicheng Liu, Yifan Qiao, Tian Xia, Yi Xu, Tony Hong, Yutong Zhou, David Sun, Shuo Yang, Yilong Zhao, Junyi Shu, Jiarong Xing, Harry Xu, Ion Stoica, Joseph Gonzalez, and Sam Kumar. 2027. Breaking the Tradeoff: Elastic and Isolated GPU Sharing with GVM. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Heraklion, Greece) *(ASPLOS '27)*. Association for Computing Machinery, New York, NY, USA, 136–152. doi:10.1145/3845814.3850071

[42] Tadaaki Mashimo, Yoshifumi Fukunishi, Narutoshi Kamiya, Yu Takano, Ikuo Fukuda, and Haruki Nakamura. 2013. Molecular Dynamics Simulations Accelerated by GPU for Biological Macromolecules with a Non-Ewald Scheme for Electrostatic Interactions. *Journal of Chemical Theory and Computation* 9, 12 (12 2013), 5599–5609. doi:10.1021/ct400342e

[43] Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam. 2012. iGPU: exception support and speculative execution on GPUs. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '12)*. IEEE Computer Society, USA, 72–83.

[44] Seung Won Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-mei Hwu. 2020. EMOGI: efficient memory-access for out-of-memory graph-traversal in GPUs. *Proc. VLDB Endow.* 14, 2 (Oct. 2020), 114–127. doi:10.14778/3425879.3425883

[45] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. arXiv:1906.00091 [cs.IR] https://arxiv.org/abs/1906.00091

[46] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. 2023. Paella: Low-latency Model Serving with Software-defined GPU Scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 595–610. doi:10.1145/3600006.3613163

[47] Marco S Nobile, Paolo Cazzaniga, Andrea Tangherloni, and Daniela Besozzi. 2017. Graphics processing units in bioinformatics, computational biology and systems biology. *Briefings in Bioinformatics* 18, 5 (09 2017), 870–885. doi:10.1093/bib/bbw058

[48] NVIDIA. 2026. cuBLAS Library. https://docs.nvidia.com/cuda/cublas/. Accessed November 2026.

[49] NVIDIA. 2026. CUDA C++ Programming Guide, Unified Memory Programming. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Accessed November 2026.

[50] NVIDIA Corporation. 2018. open-gpu-doc: Volta GV100 RAM register manual (`dev_ram.ref.txt`): channel and runlist formats. https://github.com/NVIDIA/open-gpu-doc/blob/master/manuals/volta/gv100/dev_ram.ref.txt. Accessed 2026-10-19.

[51] NVIDIA Corporation. 2026. Multi-Process Service: Architecture. https://docs.nvidia.com/deploy/mps/architecture.html. Accessed 2026-10-19.

[52] NVIDIA Corporation. 2026. NVIDIA Linux Open GPU Kernel Module Source. GitHub repository, release tag 595.91.07. https://github.com/NVIDIA/open-gpu-kernel-modules/releases/tag/595.91.07

[53] NVIDIA Corporation. 2026. Pascal Tuning Guide, Section 1.4.7: Compute Preemption. CUDA Toolkit 13.4 documentation, https://docs.nvidia.com/cuda/pascal-tuning-guide/index.html. Accessed 2026-10-22.

[54] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. 2024. CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 4236–4247. doi:10.1109/ICDE60146.2024.00323

[55] Sreepathi Pai, R. Govindarajan, and Matthew J. Thazhuthaveetil. 2014. Preemptive thread block scheduling with online structural runtime prediction for concurrent GPGPU kernels. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (Edmonton, AB, Canada) *(PACT '14)*. Association for Computing Machinery, New York, NY, USA, 483–484. doi:10.1145/2628071.2628117

[56] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2015. Chimera: Collaborative Preemption for Multitasking on a Shared GPU. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 593–606. doi:10.1145/2694344.2694346

[57] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[58] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based GPU Memory Management for Deep Learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 891–905. doi:10.1145/3373376.3378505

[59] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural support for address translation on GPUs: designing memory management units for CPU/GPUs with unified address spaces. *SIGPLAN Not.* 49, 4 (Feb. 2014), 743–758. doi:10.1145/2644865.2541942

[60] Jason Power, Mark D. Hill, and David A. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 568–578. doi:10.1109/HPCA.2014.6835965

[61] John Ravi, Tri Nguyen, Huiyang Zhou, and Michela Becchi. 2021. PILOT: a Runtime System to Manage Multi-tenant GPU Unified Memory Footprint. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 442–447. doi:10.1109/HiPC53243.2021.00063

[62] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 551–564. https://www.usenix.org/conference/atc21/presentation/ren-jie

[63] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. doi:10.1109/MICRO.2016.7783721

[64] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. arXiv:1701.06538 [cs.LG] https://arxiv.org/abs/1701.06538

[65] Weihang Shen, Yinqiu Chen, Rong Chen, and Haibo Chen. 2026. Towards Fully-fledged GPU Multitasking via Proactive Memory Scheduling. arXiv:2512.24637 [cs.OS] https://arxiv.org/abs/2512.24637

[66] Weihang Shen, Mingcong Han, Jialong Liu, Rong Chen, and Haibo Chen. 2025. XSched: Preemptive Scheduling for Diverse XPUs. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI '25)*. 671–692. https://www.usenix.org/conference/osdi25/presentation/shen-weihang

[67] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: high-throughput generative inference of large language models with a single GPU. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 1288, 23 pages.

[68] Seunghee Shin, Guilherme Cox, Mark Oskin, Gabriel H. Loh, Yan Solihin, Abhishek Bhattacharjee, and Arkaprava Basu. 2018. Scheduling Page Table Walks for Irregular GPU Applications. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 180–192. doi:10.1109/ISCA.2018.00025

[69] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 1075–1092. doi:10.1145/3627703.3629578

[70] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. 2014. Enabling preemptive multiprogramming on GPUs. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 193–204. doi:10.1109/ISCA.2014.6853208

[71] Ivan Tanasic, Isaac Gelado, Marc Jorda, Eduard Ayguade, and Nacho Navarro. 2017. Efficient exception handling support for GPUs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 109–122. doi:10.1145/3123939.3123950

[72] Shashank Verma, Wenwen Gao, Yingcan Wei, Matthias Langer, Jerry Shi, Fan Yu, Kingsley Liu, and Minseok Lee. 2022. Scaling Recommendation System Inference with NVIDIA Merlin Hierarchical Parameter Server. NVIDIA Technical Blog, https://developer.nvidia.com/blog/scaling-recommendation-system-inference-with-merlin-hierarchical-parameter-server/. Accessed 2026-10-22.

[73] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. 2016. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 161–171. doi:10.1109/ISPASS.2016.7482091

[74] Nandita Vijaykumar, Kevin Hsieh, Gennady Pekhimenko, Samira Khan, Ashish Shrestha, Saugata Ghose, Adwait Jog, Phillip B. Gibbons, and Onur Mutlu. 2016. Zorua: A holistic approach to resource virtualization in GPUs. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–14. doi:10.1109/MICRO.2016.7783718

[75] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO-52)*. Association for Computing Machinery, New York, NY, USA, 372–383. doi:10.1145/3352460.3358307

[76] Jiali Wang, Yankui Wang, Mingcong Han, and Rong Chen. 2025. Colocating ML Inference and Training with Fast GPU Memory Handover. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association, Boston, MA, 1657–1675. https://www.usenix.org/conference/atc25/presentation/wang-jiali

[77] Yidi Wang, Cong Liu, Daniel Wong, and Hyoseung Kim. 2024. GCAPS: GPU Context-Aware Preemptive Priority-based Scheduling for Real-Time Tasks. arXiv:2406.05221 [cs.DC] https://arxiv.org/abs/2406.05221

[78] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2016. Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 358–369. doi:10.1109/HPCA.2016.7446078

[79] Bo Wu, Xu Liu, Xiaobo Zhou, and Changjun Jiang. 2017. FLEP: Enabling Flexible and Efficient Preemption on GPUs. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 483–496. doi:10.1145/3037697.3037742

[80] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent GPU Sharing in Container Clouds for Deep Learning Workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 69–85. https://www.usenix.org/conference/nsdi23/presentation/wu

[81] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 595–610. https://www.usenix.org/conference/osdi18/presentation/xiao

[82] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 533–548. https://www.usenix.org/conference/osdi20/presentation/xiao

[83] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. 2016. Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 230–242. doi:10.1109/ISCA.2016.29

[84] Yechen Xu, Yifei Wang, Nathanael Ren, Yiran Chen, and Danyang Zhuo. 2026. Nixie: Efficient, Transparent Temporal Multiplexing for Consumer GPUs. In *20th USENIX Symposium on Operating Systems Design and Implementation (OSDI 26)*. USENIX Association, Seattle, WA, 2085–2101. https://www.usenix.org/conference/osdi26/presentation/xu-yechen

[85] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 Technical Report. arXiv:2412.15115 [cs.CL] https://arxiv.org/abs/2412.15115

[86] Peifeng Yu and Mosharaf Chowdhury. 2020. Fine-Grained GPU Sharing Primitives for Deep Learning Applications. In *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze (Eds.), Vol. 2. 98–111. https://proceedings.mlsys.org/paper_files/paper/2020/file/d9cd83bc91b8c36a0c7c0fcca59228f2-Paper.pdf

[87] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model Checking TLA+ Specifications. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME '99)*. Springer-Verlag, Berlin, Heidelberg, 54–66.

[88] Wei Zhang, Weihao Cui, Kaihua Fu, Quan Chen, Daniel Edward Mawhirter, Bo Wu, Chao Li, and Minyi Guo. 2019. Laius: Towards Latency Awareness and Improved Utilization of Spatial Multitasking Accelerators in Datacenters. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. Association for Computing Machinery, New York, NY, USA, 58–68. doi:10.1145/3330345.3330351

[89] Han Zhao, Weihao Cui, Quan Chen, Youtao Zhang, Yanchao Lu, Chao Li, Jingwen Leng, and Minyi Guo. 2022. Tacker: Tensor-CUDA Core Kernel Fusion for Improving the GPU Utilization while Ensuring QoS. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 800–813. doi:10.1109/HPCA53966.2022.00064

[90] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W. Keckler. 2016. Towards High Performance Paged Memory for GPUs. In *Proceedings of the 22nd IEEE International Symposium on High Performance Computer Architecture (HPCA 2016)*. 345–357. doi:10.1109/HPCA.2016.7446077