GZswap: Hardware-Managed Compressed Swap for GPU Memory Oversubscription

Boyeol Choi*
Sungkyunkwan University
Suwon, Republic of Korea
gju06051@gmail.com

Sanghyun Park
FuriosaAI
Seoul, Republic of Korea
psh2018314072@gmail.com

Sanghyun Hong
MangoBoost
Seoul, Republic of Korea
hsangh1596@gmail.com

Seokin Hong
Sungkyunkwan University
Suwon, Republic of Korea
seokin@skku.edu

Jungrae Kim
Sungkyunkwan University
Suwon, Republic of Korea
dale40@skku.edu

Abstract

Modern GPU applications are increasingly constrained by limited on-device memory capacity. Existing GPU memory expansion mechanisms, such as NVIDIA Unified Memory, extend capacity by offloading data to slower memory tiers, including CPU DRAM and CXL-attached memory. However, this approach introduces substantial overhead due to software-managed page fault handling and limited off-package link bandwidth. Consequently, the benefits of memory expansion are often limited by high-cost data migration.

This paper presents GZswap, a hardware-managed compressed swap mechanism that expands effective GPU memory capacity. GZswap reserves a portion of GPU memory as a compressed swap region. Rather than immediately evicting victim pages to secondary memory, GZswap compresses selected pages and stores them in this on-device region. Subsequent accesses to these pages can then be served through local decompression instead of off-package data migration. By offloading page fault handling to hardware and reducing off-package traffic, GZswap improves the performance of memory-oversubscribed GPU workloads.

CCS Concepts

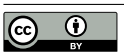
• **Computer systems organization → Heterogeneous (hybrid) systems**.

Keywords

GPU, Unified Memory, Data Compression, Zswap

ACM Reference Format:
Boyeol Choi, Sanghyun Park, Sanghyun Hong, Seokin Hong, and Jungrae Kim. 2026. GZswap: Hardware-Managed Compressed Swap for GPU Memory Oversubscription. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '26), October 19–22, 2026, Chicago, IL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3838684.3846862

*Dept. of Electrical and Computer Engineering, Sungkyunkwan University.



This work is licensed under a Creative Commons Attribution 4.0 International License.
PACT '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2915-7/2026/10
https://doi.org/10.1145/3838684.3846862

1 Introduction

Modern GPU systems are increasingly constrained by memory capacity rather than raw compute capability. Large language models, scientific simulations, and other data-intensive workloads continue to grow in memory footprint, often exceeding the capacity of on-package High Bandwidth Memory (HBM). Although HBM provides high bandwidth, its capacity scaling is limited by packaging complexity, thermal constraints, power delivery, and cost [26]. As a result, the gap between application memory demand and practical HBM capacity has become a central bottleneck in system design.

A common way to expand GPU memory capacity is *memory offloading*, which augments HBM with larger but lower-performance memory tiers such as host DRAM, CXL-attached memory [28], or storage [11]. NVIDIA Unified Memory (UM), for example, exposes a unified virtual address space and supports oversubscription by migrating data between device and secondary memory when the footprint exceeds GPU memory capacity [73].

However, offloaded data must eventually be brought back into GPU memory before it can be accessed efficiently by GPU cores. Consequently, oversubscribed execution becomes highly sensitive to migration overheads, including the latency and throughput of fault handling and the off-package link [17]. When page faults occur frequently, fault-servicing and migration costs can dominate execution time, limiting the practical benefit of memory expansion.

Prior work improves UM performance by optimizing page placement, prefetching, swap-in/out [1] scheduling, and hardware-assisted fault handling [10, 13, 18, 24, 25, 32, 40, 50, 66]. These techniques make offloading more efficient by determining which pages should reside in GPU memory, when pages should move, and how fault handling should be accelerated. Such approaches are effective when the *active working set* fits within device memory: hot pages can remain in HBM, while colder pages are placed in secondary memory.

However, when the working set itself exceeds device memory capacity, even accurate placement cannot keep all hot data resident at once. Hot pages repeatedly compete for limited HBM space, causing useful data to be evicted and later restored through the off-package path. In this regime, the dominant bottleneck is not simply the offloading policy, but the lack of effective memory capacity.

This observation motivates a different approach to GPU memory oversubscription. Rather than only optimizing how pages move

[1]This paper uses *swap-in* for CPU-to-GPU page movement and *swap-out* for GPU-to-CPU page movement.

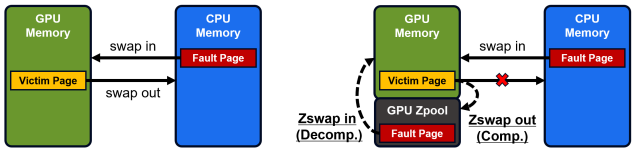


Figure 1: A comparison of existing GPU memory oversubscription mechanisms and GZswap.

between HBM and secondary memory, the system should reduce the number of useful pages that leave HBM at all. We therefore advocate compressed near-memory expansion, where eviction victims are first compressed and retained inside GPU memory before being sent to secondary memory. By storing more data within the same HBM capacity, this approach can absorb memory pressure locally more effectively, reduce off-package traffic, and make better use of on-package bandwidth during oversubscribed execution.

Figure 1 shows the high-level operation of GZswap. In conventional oversubscription, an evicted page leaves HBM and must later be restored from secondary memory. In contrast, GZswap reserves part of HBM as an on-device compressed page pool, called the *Zpool*. When a page is selected for eviction, GZswap compresses it into the Zpool. Upon reuse, GZswap restores the page through local decompression, avoiding a secondary-memory fetch and converting off-package relocation into on-device processing.

Although compressed swap has been used in CPU systems, a software-centric design such as Linux zswap [45, 78, 83] does not directly address the needs of GPU oversubscription at scale. CPU zswap relies heavily on kernel software to allocate variable-sized compressed pages, manage metadata, repeatedly reclaim Zpool space, and restore pages during latency-sensitive fault handling.

GPU applications can generate a large number of concurrent page faults during oversubscription, and performance is highly sensitive to both fault-handling throughput and I/O bandwidth. Placing these management operations on the GPU driver path would therefore increase fault-service latency and limit fault-handling throughput. To address this mismatch, GZswap integrates compressed swap into the GPU memory-management path. It provides hardware support for compressed-page metadata, Zpool management, and compressed-page swap operations, while minimizing CPU involvement on the critical fault path.

Our evaluation shows that GZswap substantially improves oversubscribed GPU execution by retaining evicted pages in compressed form within HBM. Using 20% of HBM capacity as the Zpool, GZswap improves performance by up to 59%, reduces relocation I/O traffic by up to 55%, and lowers total energy consumption by up to 29%.

This paper makes the following contributions:

- We characterize the overheads of GPU memory oversubscription and show that software fault handling and repeated offloading traffic limit performance.
- We analyze the compressibility of GPU pages and show that compressed swap can substantially expand effective device-memory capacity.
- We propose GZswap, a hardware-managed GPU-resident compressed swap layer that retains selected eviction victims on device.
- We present the GZswap microarchitecture for compressed page tracking, Zpool management, and hardware-assisted fault servicing.

2 Background

This section provides the background needed to understand GZswap by reviewing NVIDIA UM with oversubscription, the microarchitecture of GPU page-fault handling, and GPU data compression.

2.1 Unified Memory and Oversubscription

NVIDIA *Unified Memory (UM)* simplifies heterogeneous CPU–GPU programming by providing a single virtual address space across CPU and GPU memories [73]. Instead of explicit data copies, UM allows the runtime and driver to transparently manage on-demand data placement and migration. Recent NVIDIA systems extend this abstraction with *Full Unified Memory (Full-UM)*, which allows managed allocations to exceed physical GPU memory capacity [17]. Larger but slower tiers, such as host DRAM, serve as backing storage. This enables GPU *memory oversubscription*: active pages reside in high-bandwidth GPU memory when possible, while less active pages are placed in secondary memory.

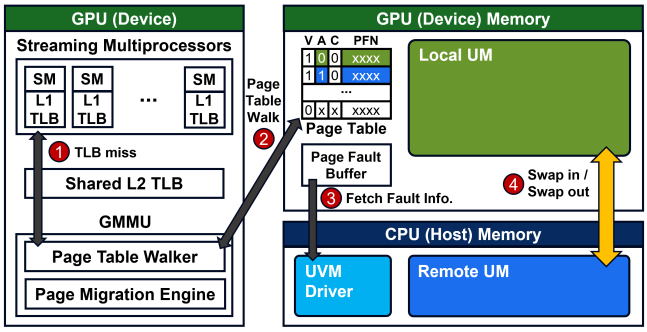
Full-UM can rely on either hardware- or software-managed coherence. Hardware-coherent Full-UM uses tightly integrated CPU–GPU platforms (e.g., NVIDIA Grace Blackwell Superchips) and coherent interconnects (e.g., NVLink-C2C) to maintain coherence across memory tiers [62, 82]. Although this can reduce software overheads, it requires specialized packaging and interconnect support, limiting its applicability to conventional discrete GPUs.

Software-coherent Full-UM is the general oversubscription mechanism for conventional discrete GPUs. It uses the GPU driver to service page faults, update mappings, and migrate pages between CPU and GPU memory domains [60]. However, under severe memory oversubscription, frequent page faults and repeated off-package migrations can dominate execution time, making performance highly sensitive to fault-servicing latency, migration throughput, and I/O bandwidth. This paper therefore targets software-coherent Full-UM and seeks to improve its performance by reducing both driver-mediated fault overhead and off-package traffic.

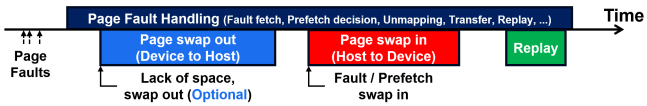
2.2 Unified Memory Microarchitecture

Software-coherent Full-UM handles oversubscription through GPU address translation and replayable page faults. Because its complete implementation is not publicly disclosed, we describe its behavior based on NVIDIA documentation and prior studies [3–5, 17, 23–25, 60, 66, 67, 71–74, 85, 86]. Figure 2 illustrates the GPU-side translation path and the corresponding page-fault handling process.

As shown in Figure 2a, a memory request from a Streaming Multiprocessor (SM) first probes the per-SM L1 TLB and then the shared L2 TLB. On a TLB miss, the request is forwarded to the GPU Memory Management Unit (GMMU) for a page-table walk. The resulting Page Table Entry (PTE) contains an Aperture flag (A), which distinguishes GPU-local and remote residency [56]. If A is zero, the translation is inserted into the TLB hierarchy and the access proceeds normally. Otherwise, the GMMU records the faulting address in a GPU-resident page-fault buffer. The host-side UVM driver fetches pending faults from this buffer in batches,



(a) Unified Memory microarchitecture.



(b) Fault handling timeline.

Figure 2: Unified Memory page-fault handling mechanism.

determines the swap-in region, updates the mappings, and issues migration requests through the GPU Page Migration Engine (PME).

Modern UM systems use prefetching to amortize migration overhead. A 4 KB fault may trigger migration of a 64 KB leaf containing sixteen contiguous pages, while tree-based neighborhood prefetching [24] can aggregate multiple leaves within a 2 MB region. Larger transfers amortize fixed fault-handling and DMA costs and can reduce subsequent faults when neighboring pages exhibit spatial locality. However, coarser migration can increase overfetch, consuming transfer bandwidth and device-memory capacity for data that may not be accessed. Thus, UM varies swap-in granularity from 64 KB to 2 MB to balance transfer efficiency and overfetch.

Figure 2b shows the corresponding fault-handling timeline. After detecting a fault, the UVM driver checks whether sufficient GPU memory is available for the incoming data. If GPU memory is full, it first swaps out victim data to free space before migrating the faulting data into GPU memory. Page-fault handling can therefore involve a serialized sequence of swap-out and swap-in operations. After migration completes and the mapping is updated, the driver triggers replay so that the stalled GPU access can proceed. Under high memory pressure, however, this sequence occurs more frequently, increasing fault-service overhead and repeated off-package data movement, and ultimately degrading performance.

2.3 GPU Data Compression

GPU data compression has been explored in two main directions. The first targets bulk, application-managed data streams. NVIDIA's nvCOMP library [63] provides GPU-accelerated compression and decompression for formats such as LZ4 [51], Snappy [27], and Deflate [22]. Applications can transfer compressed datasets to the GPU and decompress them on device before processing, reducing storage and data-transfer overheads for bulk data movement. Recent GPU architectures, such as NVIDIA Blackwell, also provide dedicated hardware support for decompression [61].

However, stream-oriented compression does not naturally fit transparent page migration. A software-based approach would place compression and decompression on the page-fault path, requiring GPU kernel launches, context switches, and synchronization on every fault before execution can resume. Existing hardware support also primarily targets stream decompression and lacks integration with page-fault handling.

The second direction uses transparent hardware-based compression to increase effective capacity and memory bandwidth [7, 9, 19, 20, 33, 36, 42, 49, 55, 69]. Because these mechanisms operate on latency-critical cache and memory accesses, they typically use lightweight compression at cacheline or sector granularity to minimize normal access overheads [46]. For example, Buddy Compression [20] compresses each 128 B cacheline to a fixed target size and spills the remainder into host memory. Its efficiency therefore hinges on how consistently cachelines meet the target compression ratio: a line that compresses below the target leaves reserved space unused, while one that exceeds it is re-fetched remotely on every access, consuming scarce off-chip I/O bandwidth.

GZswap sidesteps this trade-off by compressing whole pages on the page-movement path and storing them at variable size. A poorly compressible block can be hidden by well-compressed blocks within the same page, and the already time-consuming migration can accommodate stronger algorithms. Poorly compressible data primarily consume additional capacity rather than generating fine-grained remote accesses, allowing GZswap to avoid a fixed compression target. Furthermore, GZswap builds on UM to leverage working sets smaller than the total footprint.

3 Related Work

This section reviews two lines of work related to GZswap: GPU UM management and CPU compressed swapping. These directions address complementary aspects of memory oversubscription by reducing data movement and increasing effective memory capacity.

3.1 Unified Memory Management

Prior work on GPU UM oversubscription primarily improves data placement and movement through adaptive prefetching and eviction policies, or reduces the cost of page-fault handling.

Interplay [24] proposes locality-aware eviction to reduce unnecessary migration, while Adaptive Page Migration [25] adapts migration and zero-copy decisions for irregular applications. Griffin [13] improves page migration and scheduling in multi-GPU systems, SUV [10] uses compiler-guided placement and prefetching, and Forest [50] configures prefetching based on access behavior. Together, these techniques use runtime or compiler-derived access information to improve residency and movement decisions, reducing unnecessary data movement under memory pressure.

Other approaches focus on reducing the cost of accessing nonresident GPU pages. Toward UM [86] uses a far-fault MSHR to tolerate outstanding page faults and reduce SM stalls. UVMMU [66] provides hardware-assisted UM support that reduces software involvement in address translation, fault handling, and page migration. DREAM [54] instead enables efficient GPU-driven page migration through an RDMA-capable device, avoiding CPU/OS involvement in the migration path. Together, these techniques reduce remote-access overhead by tolerating page-fault latency or directly offloading memory-management operations from the CPU.

However, when the active working set exceeds device-memory capacity, reusable hot pages still compete for limited GPU memory space, causing recurring off-package traffic. GZswap complements these techniques by increasing effective device-memory capacity rather than optimizing page movement or fault handling.

3.2 Compressed Swap in CPUs

Linux *zswap* is a compressed cache that stores evicted pages in a memory-resident compressed pool before writing them to backing storage, reducing swap I/O at the cost of compression and memory-access overheads [78]. Software compressed-swap systems often use LZ/Deflate-style dictionary-based byte-stream algorithms, which provide high compression ratios but operate at coarse granularity and consume substantial CPU cycles and memory bandwidth during compression and decompression [31, 35, 68].

The high cost of DRAM in large-scale datacenters has motivated compressed memory and memory tiering to reduce memory capacity costs. Google's Software-Defined Far Memory [45] compresses cold pages to create a software-managed far-memory tier, while Meta's TMO [83] transparently offloads cold data to cheaper tiers, including compressed memory. More recently, TierScape [44] explores multiple compressed tiers with different capacity-performance trade-offs. Compressed swap is also useful in memory-constrained mobile systems. eZswap [41] improves memory efficiency by selectively compressing and caching beneficial pages, while related smartphone memory expansion techniques reduce local-memory pressure through remote offloading [48].

To further reduce the processing overhead of compressed memory, hardware acceleration has also been explored for compression and decompression. Dedicated compression engines in IBM POWER9/z15 and Intel IAA/QAT-based designs reduce CPU involvement in data compression [1, 34, 84]. TMCC [65] develops a hardware Deflate engine for 4 KB page compression to reduce the compute overhead of software-based compression.

Overall, CPU compressed-memory systems trade compute and memory bandwidth for reduced swap traffic. However, their management remains centered on CPU-side kernel mechanisms for allocation, metadata, reclamation, and fault handling. In contrast, GZswap integrates compressed-page management into the GPU page-fault path, using hardware to service page faults and manage the on-device Zpool. Thus, GZswap adapts compressed swap to GPU memory oversubscription by redesigning the management path rather than simply accelerating compression.

4 Motivation

This section first examines the compression opportunity in GPU memory and its potential for increasing effective device-memory capacity. We then characterize GPU memory oversubscription and show that frequent page migration and off-package traffic substantially degrade performance.

4.1 Compression Opportunity

As discussed in Sections 2.3 and 3.2, compression granularity presents an important trade-off. Fine-grained cacheline compression favors low access latency but exposes a limited compression window, whereas page-granular compression can exploit redundancy across a larger region at higher processing cost. Since compressed swap operates on the page-movement path rather than the normal memory-access path, page-level compression can provide a larger capacity benefit without adding decompression to every memory access.

To quantify this opportunity, we analyze GPU memory compressibility using representative cacheline compression schemes, including FPC, CPack, BDI, SC2, GBDI, and BPC [2, 6, 8, 16, 42, 69], together with page-granular Deflate [1, 65]. We extend NVIDIA's binary instrumentation tool, NVBit [80], to collect GPU memory dumps during execution. Across 57 diverse GPU workloads [14, 15, 29, 38, 53, 75, 81], we periodically capture 10 memory snapshots per workload to reflect compression behavior across execution phases. We exclude all-zero pages to avoid artificially inflating compression ratios and evaluate the schemes on realistic GPU memory contents.

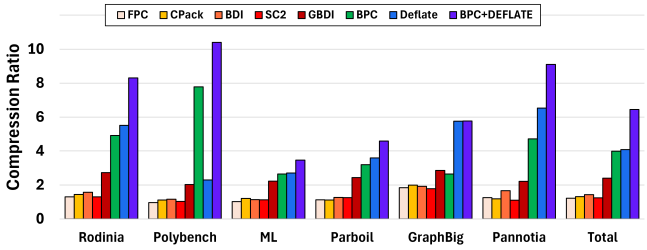


Figure 3: Compression ratio of GPU memory snapshots.

Figure 3 summarizes the results. Most cacheline compression schemes, including FPC, CPack, BDI, SC2, and GBDI, achieve compression ratios of 1–2.5×, as their compression windows expose limited redundancy. In contrast, page-granular Deflate exploits redundancy across a larger region and achieves 4.07× on average.

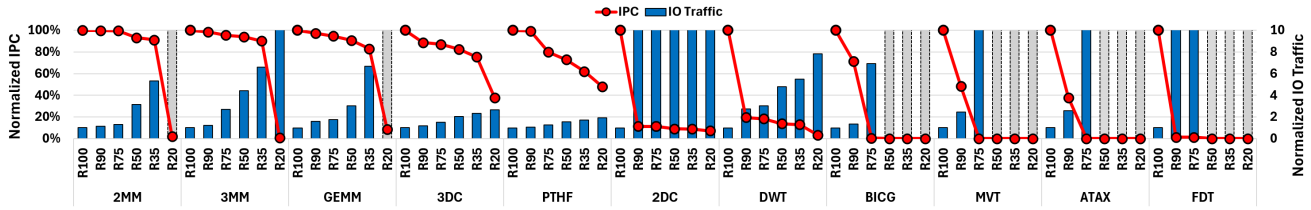
BPC is an important exception among lightweight schemes, achieving 3.98× on average despite its fine-grained operation.[2] GPU applications often use structure-of-arrays layouts with strong similarity among adjacent values [47], which matches BPC's bit-plane representation well. These results indicate that compression effectiveness depends not only on granularity, but also on how well the compression algorithm captures GPU data patterns.

BPC and Deflate therefore provide complementary strengths. BPC efficiently captures regular GPU data patterns with lightweight compression, while Deflate exploits broader redundancy at page granularity. GZswap combines these properties by selecting between BPC and Deflate for each page, achieving an average compression ratio of 6.45×. This substantial compressibility motivates retaining eviction victims in compressed form within device memory. We next examine the oversubscription overhead that such compressed residency can reduce.

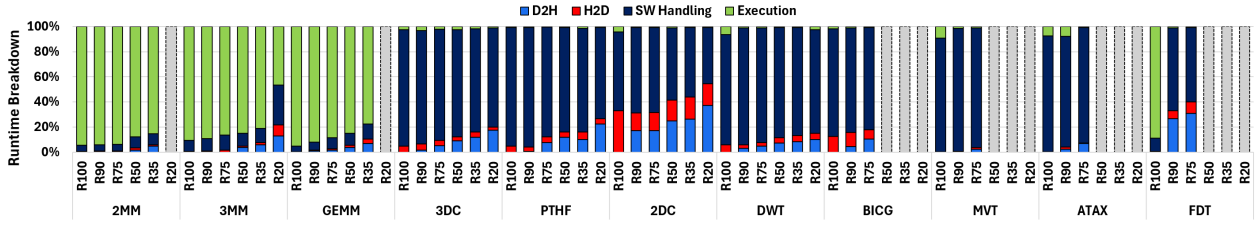
4.2 Memory Oversubscription Overheads

Experiment Setup. Our platform consists of an NVIDIA RTX 3090 GPU [76] and an AMD EPYC 7313 CPU [77] connected through PCIe Gen4 ×16, running CUDA 12.3 with NVIDIA driver 545.23. We use

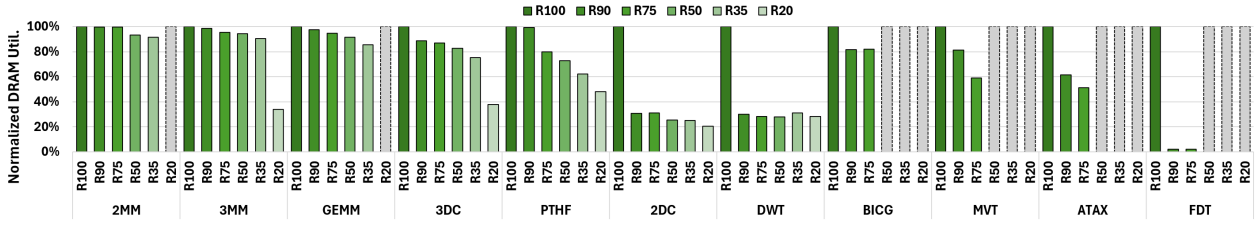
[2]BPC performs particularly well on PolyBench [29]. As a performance-oriented kernel benchmark, PolyBench commonly generates dense-array inputs using simple deterministic formulas rather than application-derived datasets, resulting in highly regular values with small adjacent deltas and low-entropy bit-planes.



(a) Performance and I/O traffic under reduced device memory capacity.



(b) Execution time breakdown.



(c) Memory bandwidth utilization.

Figure 4: Effect of shrinking near-memory capacity on relocation traffic, execution time, and memory bandwidth utilization. Gray bars denote profiling failures caused by excessively long execution time.

UVM workloads from UVMBench [30]. To avoid distortion[3] from small program sizes, we include only workloads whose memory footprints are at least 1 GB. We use NVIDIA profilers to collect page-fault behavior, data transfers, and GPU memory bandwidth utilization [58, 59], capturing key oversubscription overheads.

We denote the effective device-memory availability using an *R prefix* [32]. R is the ratio of available GPU memory to the application memory footprint for each workload. For example, R100 indicates that the full footprint fits in GPU memory, corresponding to no oversubscription. In contrast, R50 means that only 50% of the footprint can be resident in GPU memory; for a 10 GB footprint, this corresponds to 5 GB of available GPU memory capacity.

For each workload, we first measure its memory footprint and then emulate the target R value using dummy GPU allocations, following NVIDIA's methodology [17]. We vary R to control the degree of oversubscription and study how reduced near-memory capacity affects performance, traffic, and GPU memory utilization.

IPC and I/O Traffic. Figure 4a shows the impact of shrinking near-memory capacity on IPC and offloading traffic. Performance degrades at different R values across benchmarks. Across workloads, IPC remains stable until R reaches a workload-specific threshold; after this point, offloading traffic increases sharply and IPC drops rapidly. For example, 3MM maintains about 90% performance until R20, after which IPC drops sharply. In contrast, BICG reaches this transition earlier and shows a steep performance drop around R75. Some workloads are even more sensitive: 2DC, DWT, and FDT show severe performance degradation even with a small reduction in device memory relative to their footprints.

This threshold behavior is determined by each program's active working-set size [21]. When effective device-memory capacity is larger than the active working set, UM policies such as LRU-based eviction can keep most frequently used pages in GPU memory, and the workload maintains performance close to the no-oversubscription case. However, once the working set exceeds the available capacity, even a reasonable eviction policy must evict pages that will soon be reused. The runtime then repeatedly restores demanded pages and evicts other hot pages, creating a thrashing regime. As a result, execution becomes increasingly dominated by the offloading path rather than useful GPU computation.

Overhead Breakdown. To identify the sources of overhead, Figure 4b breaks down kernel execution time using timeline information from Nsight Systems traces. For each interval, we classify time into four components: H2D (CPU→GPU) transfer, D2H (GPU→CPU) transfer, residual software-handling time, and remaining GPU execution time. H2D and D2H capture CPU-to-GPU swap-in and GPU-to-CPU swap-out traffic, respectively. Because both transfers are generally serialized during page-fault handling rather than fully overlapped[4], we account for them as separate transfer components. The residual software component captures page-fault handling time not directly attributed to transfer, including fault

[3]Small-footprint benchmarks can be dominated by fixed runtime overheads. In our setup, the CUDA runtime occupies approximately 286 MB, which can significantly affect the memory pressure of small workloads.

[4]H2D–D2H overlap is rarely observed except in workloads with bursty page faults, such as MVT and ATAX, where multiple 2 MB-region faults are serviced concurrently. Even then, the overlap accounts for less than 5% of total profiled time.

processing, mapping updates, and replay management [3, 5]. Remaining time is classified as GPU execution.

The breakdown shows that software handling is a significant overhead, motivating dedicated hardware fault handling. However, as oversubscription intensifies, H2D and D2H transfers also account for a large fraction of execution time. Thus, hardware fault handling alone is insufficient; an effective solution must also reduce off-package traffic over the bandwidth-limited interconnect.

Memory Bandwidth Utilization. Figure 4c shows that GPU DRAM bandwidth utilization often decreases as oversubscription becomes more severe. This may seem counterintuitive because higher memory pressure causes more page migration. However, much of this traffic uses the off-device I/O path rather than regular GPU memory accesses. While the GPU waits for fault handling and secondary-memory transfers, fewer active warps can issue requests to device DRAM. Consequently, the offloading path becomes the bottleneck while on-device memory bandwidth remains underutilized. This underutilization reveals an opportunity for GZswap. If part of the offloading traffic can be converted into local compression, decompression, and device memory accesses, the system can use otherwise idle device-memory bandwidth while reducing pressure on the slow off-package I/O path during oversubscribed execution.

Overall. Together, these observations motivate GZswap to combine hardware-managed fault handling with compressed residency. By retaining evicted data in compressed form, GZswap converts off-package transfers into local compression, decompression, and device-memory accesses, exploiting otherwise underutilized device-memory bandwidth.

5 GZswap

We propose *GZswap*, a hardware-managed compressed swap mechanism for GPU memory oversubscription. GZswap reserves part of GPU memory as an on-device compressed swap space, allowing selected victim pages to remain on device rather than being immediately evicted to secondary memory.

5.1 Overview

Figure 5 illustrates GZswap's three-layer memory organization. *Local-UM* contains uncompressed pages resident in GPU memory, whereas *Remote-UM* contains uncompressed pages stored in secondary memory. The *Zpool* is a reserved region of GPU memory that stores compressed pages as an intermediate tier. Together, these layers allow pages in the Zpool to remain on device without being directly accessible to GPU cores.

When memory pressure requires additional free Local-UM space, GZswap compresses selected victim pages and places them in the on-device Zpool, freeing their original space. We refer to this operation as *zswap-out*. If a compressed page is later accessed, GZswap performs *zswap-in*: it locates the corresponding compressed data in the Zpool, restores the page to Local-UM, and resumes the faulting access. This local restoration avoids off-package data movement whenever the requested page remains in the Zpool.

To support this memory hierarchy, GZswap slightly extends the GPU page table. In the baseline GPU page table, the Aperture flag (A) in each Page Table Entry (PTE) indicates whether a page resides in Local-UM or Remote-UM [56]. GZswap adds a Compression flag

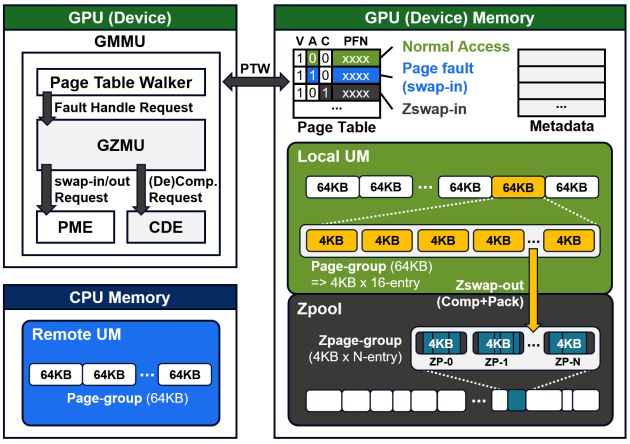
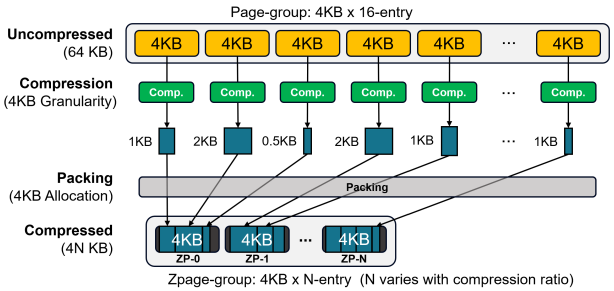


Figure 5: GZswap architecture overview.

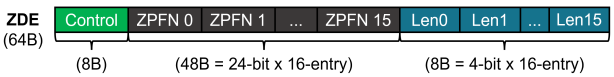
(C) to indicate whether the page is stored in compressed form in the Zpool. Together, these flags distinguish among the three memory layers: Local-UM, Remote-UM, and Zpool. For Zpool-resident pages, the physical address field is repurposed to locate the corresponding compression metadata (Section 5.2).

5.2 Page Compression and Packing

Although GZswap preserves 4 KB page granularity for memory accesses, it uses 64 KB page-groups as the unit of migration and swap management. This granularity matches NVIDIA UM's 64 KB speculative prefetch unit, while conventional UM may further aggregate multiple 64 KB regions into prefetches of up to 2 MB [5, 24, 73]. The 64 KB unit preserves transfer efficiency by amortizing fixed costs such as DMA setup. In software-managed UM, coarser prefetching further amortizes driver and migration-scheduling overheads; because GZswap offloads fault handling to hardware, it does not rely on such coarse aggregation. This allows GZswap to retain 64 KB transfer efficiency while limiting overfetch from larger migration units. Each 64 KB page-group consists of sixteen contiguous 4 KB pages. In Local-UM, these are ordinary physical pages, while in the Zpool the same 4 KB unit serves as a *zpage*, the allocation unit for compressed data. A compressed page-group may occupy a variable number of zpages depending on its compressed size.



(a) Organization of zswap-out pages.



(b) Zswap descriptor entry.

Figure 6: Page packing and metadata organization.

Figure 6a illustrates the compression and packing process. GZswap compresses each 4 KB page in the victim page-group independently. For each page, the compression engine evaluates BPC and Deflate in parallel and selects the smaller compressed output. The resulting compressed byte streams are then concatenated and packed into a sequence of 4 KB zpages. Thus, the actual number of zpages occupied by a compressed 64 KB page-group after packing depends on its compression ratio, with up to sixteen zpages used; if all sixteen are required, the data is stored uncompressed.

GZswap records packed page-groups in the *Zswap Descriptor Table (ZDT)*, with one *Zswap Descriptor Entry (ZDE)* per compressed 64 KB page-group. As shown in Figure 6b, each ZDE records per-page status and compression algorithms, Zpage Physical Frame Numbers (ZPFNs) identifying zpages containing compressed data, and encoded lengths specifying compressed sizes. These fields allow GZswap to reconstruct the layout and locate compressed pages.

The ZDT resides in GPU memory and is indexed by a PTE offset. During zswap-in, GZswap uses this offset to locate the ZDE, reads zpage locations and per-page encoded lengths, and reconstructs the compressed byte range of the requested 4 KB page. This enables critical-page-first restoration: the faulting page is decompressed first so the stalled access can resume quickly. The remaining pages in the 64 KB page-group are restored in the background, preserving group-level migration without delaying the faulting access.

This 4 KB compression granularity balances compression efficiency and fault latency: compared with cacheline compression, it provides a larger compression window and supports packing across cacheline boundaries. At the same time, independently compressing the sixteen pages allows the faulting page to be restored without decompressing the entire 64 KB page-group.

Chunk granularity. A chunk is the fixed-size unit used to encode each page's compressed size in a ZDE; the encoded length records the number of chunks occupied by the page. Chunk size trades packing efficiency for metadata overhead.

Based on the compression measurements in Section 4.1, 64, 128, 256, and 512 B chunks yield compressed capacities of 82.7%, 82.1%, 80.1%, and 72.7%, respectively. Smaller chunks improve capacity but require more length metadata, whereas larger chunks incur greater rounding loss. GZswap therefore uses 256 B chunks, retaining most of the capacity benefit while keeping metadata overhead modest.

5.3 Hardware-based Page Fault Handling

GZswap offloads fault handling to hardware to reduce fault-service latency and improve migration throughput. This follows the hardware-based fault-handling approach of prior work [66], but extends it with support for compressed swap.

As shown in Figure 5, GZswap augments the GPU memory-management path with two hardware components: the GPU Zswap Management Unit (GZMU) and the compression/decompression engine (CDE). The GZMU is embedded in the GPU Memory Management Unit (GMMU) and coordinates page-fault handling, compressed-page metadata lookup, and available-region management. The CDE performs page compression and decompression using the hybrid BPC–Deflate compression scheme. Together, these components reduce the two dominant costs of memory oversubscription: software-managed fault servicing and off-package I/O traffic.

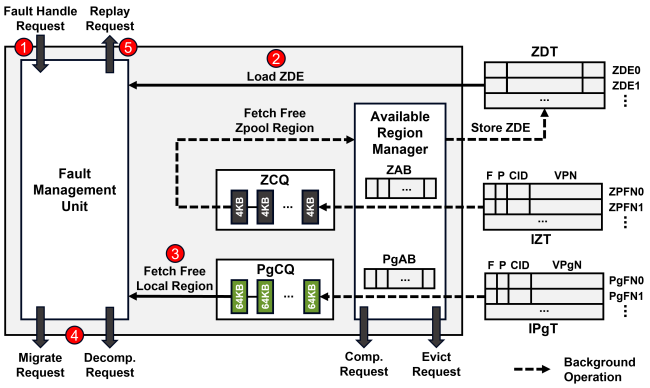


Figure 7: Microarchitecture of the GPU Zswap Management Unit, with the ZDT, IZT, and IPgT allocated in GPU memory.

As in modern GPUs, a TLB miss invokes the GMMU page-table walker [73]. When the page-table walker reaches a leaf PTE with either the A or C flag set, indicating Remote-UM or Zpool residency, respectively, the GMMU forwards the fault request to GZMU rather than returning a valid translation to the SM. The forwarded request includes the residency flags and a source field that identifies a Remote-UM PFN for an A-flagged PTE or a ZDE offset for a C-flagged PTE. GZMU uses this information to select hardware-managed migration or device-local decompression, respectively. GZswap does not require modifications to existing page-walk caches or the TLB hierarchy. GZswap-specific handling begins only after the page-table walker reaches the leaf PTE and detects either residency flag. After restoring the faulting page into Local-UM, the FMU updates the PTE with a conventional Local-UM mapping, so subsequent translations proceed through the existing hierarchy without GZswap-specific interpretation.

Both Remote-UM and Zpool faults share the same high-level requirement: *GZswap must prepare a 64 KB destination region in Local-UM and restore the source data in an uncompressed form accessible to GPU cores.* The GZMU supports this process through two components: the Fault Management Unit (FMU), which services both fault types, and the Available Region Manager (ARM), which prepares free space in Local-UM and the Zpool.

Fault Management Unit. Figure 7 highlights the FMU fault-handling path within the GZMU. **(1)** When a page-table walk reaches a nonresident PTE, the GMMU forwards the fault request to the FMU. The request carries the residency type and the source field from the PTE. **(2)** For a Zpool fault, the FMU uses the ZDE offset in the source field to fetch the corresponding ZDE from the ZDT. Remote-UM faults bypass this lookup because the source field directly identifies the source PFN in Remote-UM. **(3)** The FMU obtains a free 64 KB destination region in Local-UM from the Page-group Candidate Queue (PgCQ), which is maintained by the ARM. **(4)** Based on the fault type, the FMU issues either a migration request to the Page Migration Engine (PME) for a Remote-UM fault or a decompression request to the CDE for a Zpool fault. **(5)** Under a *critical-page-first* policy, the FMU updates the mapping and replays the stalled access as soon as the faulting 4 KB page is restored in Local-UM; the remaining pages in the page-group are restored in the background.

As described in Section 5.2, the critical-page-first policy applies to both fault types. For Remote-UM faults, the PME prioritizes the faulting 4 KB page, while for Zpool faults, the ZDE locates its compressed byte range so that the CDE can decompress it first, reducing overall fault-service latency.

Available Region Manager. The ARM prepares free Local-UM and Zpool regions for page-fault handling, zswap-out, and space reclamation. Without such preparation, a fault could stall while searching for free space. To track region ownership, GZswap maintains two reverse-mapping structures: the Inverted Page-group Table (IPgT) and the Inverted Zpage Table (IZT). Inspired by inverted page tables [70], IPgT maps each 64 KB Local-UM page-group frame to its virtual page-group, while IZT maps each 4 KB Zpool zpage to the compressed page-group that owns it. These mappings allow ARM to find free regions, select victim page-groups, and reclaim Zpool space without CPU-side software management.

ARM keeps allocation off the critical path using two candidate queues. The Page-group Candidate Queue (PgCQ) stores free 64 KB Local-UM page-groups and is consumed by the FMU for Remote-UM or Zpool faults. The Zpage Candidate Queue (ZCQ) stores free 4 KB zpages and is consumed when zswap-out packs compressed data into the Zpool. ARM refills both queues in the background to keep free destination regions readily available.

To avoid full-table scans during queue refill, ARM maintains compact availability bitmaps that summarize groups of reverse-mapping entries. The Page-group Availability Bitmap (PgAB) summarizes the IPgT: each PgAB bit represents 32 consecutive 64 KB Local-UM page-group frames and indicates whether the region contains at least one free page-group. Similarly, the Zpage Availability Bitmap (ZAB) summarizes the IZT: each ZAB bit represents 256 consecutive 4 KB Zpool zpages and indicates whether the region contains available zpages. When PgCQ or ZCQ requires refill, ARM first scans the corresponding bitmap and then accesses only the associated reverse-mapping entries needed for allocation or reclamation.

When PgCQ cannot be refilled from free Local-UM regions, ARM selects a victim page-group using the IPgT and sends it to the CDE. The compressed output is packed into zpages from ZCQ and recorded in the ZDT, after which the original Local-UM region is released to PgCQ. If the Zpool lacks sufficient free zpages, our implementation reclaims space by selecting a compressed 64 KB page-group, decompressing it through the CDE, and evicting the reconstructed page-group to Remote-UM through the conventional UM path. The released zpages are then returned to ZCQ.

If decompression throughput becomes a bottleneck, ARM could instead bypass zswap-out and directly evict a Local-UM page-group to Remote-UM. GZswap keeps Remote-UM pages uncompressed to preserve CPU accessibility and conventional UM semantics. Storing compressed pages in Remote-UM could further reduce off-package traffic, but would require additional host-side compressed-page management and decompression support.

Metadata Access Frequency. Prior compressed-memory systems show that metadata accesses can introduce significant overhead because compressed-address translation and variable-size placement may add memory accesses on the critical path [19, 20, 65]. Because GZswap preserves normal 4 KB memory accesses for Local-UM pages, its metadata structures, including the ZDT, IPgT, IZT, and availability bitmaps, are outside the regular access path and consulted only during GZMU-managed page transitions. The primary fault-path metadata access is a ZDE lookup, where one 64 B descriptor represents an entire 64 KB compressed page-group. Other metadata structures are accessed mainly for allocation, reverse mapping, and Zpool reclamation rather than for ordinary GPU memory references. Thus, metadata traffic scales primarily with page-management activity instead of the much more frequent regular memory accesses. We quantify this overhead using the simulation setup described in Section 6.1.

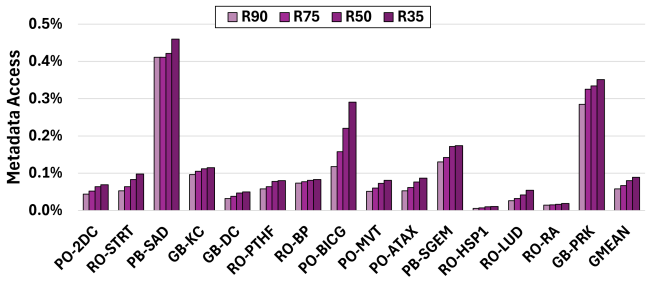


Figure 8: Metadata accesses as a fraction of total device-memory accesses across memory-pressure levels.

Figure 8 shows that metadata accesses generally increase with memory pressure as page transitions become more frequent. Even at R35, the most severe configuration evaluated, metadata accesses remain below 0.5% of total device-memory accesses across all workloads and below 0.1% for most workloads. The increase under stronger memory pressure therefore reflects more frequent GZMU-managed transitions rather than metadata accesses on the regular memory path. Moreover, the consistently small fraction indicates that metadata traffic remains minor relative to application and page-movement traffic. Thus, GZswap's metadata management remains lightweight even under severe memory pressure.

5.4 Hardware Overhead

We quantify GZswap's overheads using an NVIDIA A100-class GPU [57] with 80 GB of HBM and an 826 mm^2 die. Unlike the default evaluation, which uses a 20% active Zpool, we conservatively provision metadata for a Zpool spanning the full device-memory address range, providing an upper-bound estimate across Zpool ratios and avoiding underestimation under larger Zpool configurations.

Table 1: GZswap overhead summary.

Component	Quantity	Overhead
Metadata Overhead		
IPgT	1.25M entries @ 8 B	10 MB (0.013% of 80 GB)
IZT	20M entries @ 8 B	160 MB (0.20% of 80 GB)
ZDT	20M entries @ 64 B	1,280 MB (1.60% of 80 GB)
Total Capacity	–	**1.81% of 80 GB**
Hardware Area Overhead		
CDE	1/4 HBM BW engine	0.255% of 826 mm^2
Avail. Bitmaps	SRAM buffers	0.0035% of 826 mm^2
Total Area	–	**≈0.26% of die size**

Table 1 summarizes the results. For metadata, IPgT uses 8 B entries at 64 KB granularity and IZT uses 8 B entries at 4 KB granularity to track physical device memory, so both structures scale with the number of physical frames. ZDT stores one descriptor

per compressed page-group, but must be provisioned assuming the best-case compression ratio (16×) to support the maximum number of compressed page-groups that can reside in the Zpool. Under this assumption, a 64 KB page-group can occupy a single 4 KB zpage, requiring ZDT capacity equivalent to one 64 B descriptor per zpage and capturing the maximum descriptor demand. Overall, GZswap metadata occupies 1.81% of HBM even under this conservative provisioning. Since all three structures scale with physical frames, their relative overhead remains constant with device-memory capacity.

The dominant hardware area overhead comes from the CDE. Provisioning the CDE for one quarter of peak HBM bandwidth corresponds to 0.255% of an A100 die in an ASAP 7 nm process [42, 65]. The availability bitmaps, modeled using CACTI 7.0 [12], contribute only 0.0035% of die area. Overall, the total area overhead of GZswap is approximately 0.26% of GPU die size, which is modest relative to the effective-capacity gain and relocation reduction it enables.

6 Evaluation

This section describes our simulation setup and baselines, and evaluates GZswap in terms of system performance, I/O traffic, and energy consumption. We further compare GZswap against Buddy Compression and examine sensitivity to Zpool capacity.

6.1 Evaluation Setup

We implement GZswap in a trace-driven, cycle-accurate GPU simulation framework by integrating Accel-Sim, UVMSmart, and Ramulator [24, 39, 43]. Following prior UM studies [32], we model an NVIDIA A100-class [57] GPU scaled to one-fifth of its original configuration to make oversubscription experiments tractable across memory-pressure levels [5] while preserving the relative behavior of the memory hierarchy and interconnect. Table 2 summarizes the simulated system configuration, including the compression/decompression engine [42, 65] used by GZswap.

Table 2: Simulated system configuration.

Field	Value
Target GPU	NVIDIA A100-class GPU (1/5 scale-down)
SMs	21 SMs, 64 warps/SM, 65536 regs/SM, 901 MHz
L1 cache / Shared memory	192 KiB/SM, 128 B line (32 B sectors), LRU
L2 cache	8 MiB, 16-way, 128 B line (32 B sectors), LRU
Device memory	8 channels, 128-bit bus, 2 KiB row buffer, 8 dies, 4 bank groups/channel, 4 banks/bank group, FR-FCFS, 1 GHz, 256 GB/s peak BW
DRAM timing	CL=14, RCD=14, RAS=33, WR=16, RP=14
PCIe bandwidth	12.8 GB/s (1/5 of PCIe 4.0 ×16)
Page-fault handling latency	Software: 26.23 µs Hardware: 20 GPU cycles + 8.23 µs unmapping
Zpool ratio [78]	20% of device memory
Compression / decompression engine [42, 65]	Provisioned at 1/4 of peak device-memory BW Compression: 662 ns/page Decompression: 277 ns/page (140 ns/half-page)

Page-fault latency is parameterized using Nsight Systems and Linux `perf` profiling on the platform in Section 4.2. The measured software-handled fault latency is 26.23 µs under software-managed UVM, including 8.23 µs for CPU–GPU unmapping in the UVM batch-handler and driver path. For hardware-assisted fault handling, we model a 20-cycle GPU-side handler while retaining the same 8.23 µs host-side unmapping cost [59, 79].

Our end-to-end simulation models the latency and finite throughput of the CDE, together with metadata accesses and the HBM traffic incurred by compression and decompression. Compression and decompression requests contend for the modeled CDE bandwidth under concurrent page faults, so the simulator does not assume unlimited processing throughput. The simulator also models Zpool reclamation under sustained memory pressure when insufficient free zpages are available, including decompression of compressed page-groups and their subsequent migration to Remote-UM.

By default, GZswap reserves 20% of device memory as the Zpool, following the default maximum pool size used by Linux *zswap* [78]. This reservation does not increase the total device-memory budget relative to the baselines. For example, when a baseline is evaluated with 10 GB of available GPU memory, GZswap is evaluated with the same 10 GB budget, partitioned into 8 GB of normal Local-UM and 2 GB of Zpool. We further evaluate Zpool-ratio sensitivity across workloads and memory-pressure levels in Section 6.6.

We compare GZswap with three UM mechanisms. **2MBE** [86] models NVIDIA's UM with tree-based neighborhood prefetching, 2 MB eviction, and software fault handling. **TBNE** [24] strengthens the software baseline with tree-based neighborhood eviction. **UVMMU** [66] models hardware-assisted 64 KB page migration, but still evicts pages to far memory. **GZswap** uses the same 64 KB migration granularity and pseudo-random victim selection as UVMMU, but replaces far-memory eviction with compressed on-device residency. This separates the benefit of hardware fault handling from the benefit of memory expansion. We additionally compare GZswap with **Buddy Compression** [20] in Section 6.5 to position GZswap against prior GPU compression-based memory expansion.

We evaluate workloads from four representative benchmark suites [15, 29, 53, 75] with diverse access characteristics and memory footprints. Even with the 1/5 scale-down, cycle-level oversubscription simulation remains computationally expensive; therefore, we exclude workloads with impractical simulation times to maximize benchmark coverage within a practical simulation budget.

Table 3: Evaluated benchmarks with acronyms.

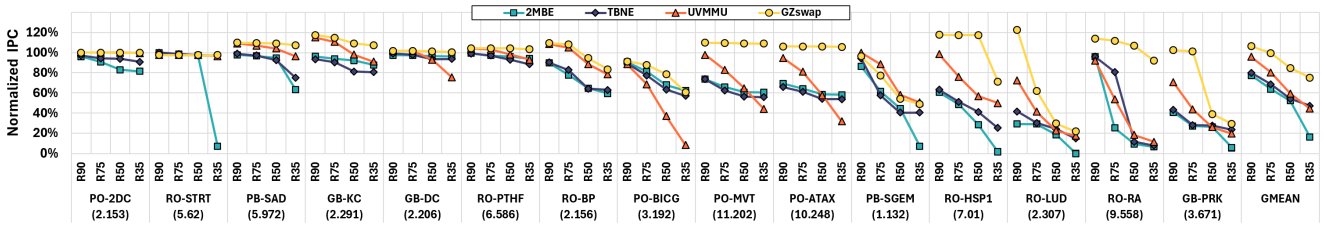
Suite	Workloads
PolyBench (PO)	2dconv (2DC), bicg (BICG), mvt (MVT), atax (ATAX)
GraphBIG (GB)	pagerank (PRK), degree centrality (DC), k-core (KC)
Parboil (PB)	sad (SAD), sgemm (SGEM)
Rodinia (RO)	backprop (BP), lud (LUD), streamtriad (STRT), hotspot-1d (HSP1), pathfinder (PTHF), ra (RA)

Table 3 lists the evaluated workloads and acronyms. We evaluate all mechanisms under the same R-notation setup used in Section 4.2, where R denotes the ratio of available device memory to each workload's memory footprint. We vary R by reducing available device-memory capacity relative to that footprint and normalize all results to TBNE at R100 unless otherwise noted.

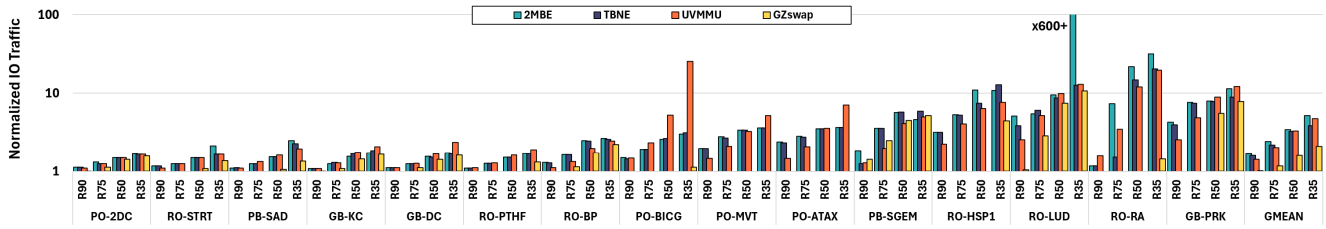
6.2 System Performance

Figure 9a shows system performance as available device memory decreases from R100 to R35, normalized to TBNE at R100, together with the per-benchmark effective Zpool compression ratios. As

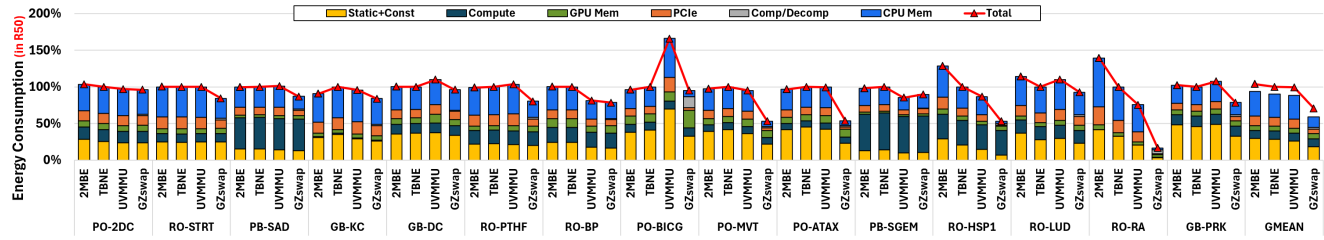
[5] Even with downscaling, cycle-level oversubscription simulation can take several weeks for a single workload. Simulating a full A100 would require proportionally larger problem sizes to avoid much of the footprint fitting in the larger L2 cache, further increasing simulation time.



(a) IPC normalized to TBNE at R100 under different memory-pressure levels, with per-benchmark Zpool compression ratios.



(b) I/O traffic normalized to TBNE at R100 under different memory-pressure levels.



(c) Energy consumption breakdown normalized to TBNE at R50.

Figure 9: Overall evaluation of GZswap: (a) system performance with per-benchmark effective Zpool compression ratios after packing, (b) relocation I/O traffic, and (c) energy consumption. Energy results are shown only at R50 for readability.

memory pressure increases, all baselines suffer from oversubscription overhead, but the magnitude of degradation depends strongly on each workload's working-set behavior and access pattern.

Compared with 2MBE and TBNE, GZswap benefits from both hardware-managed fault handling and compressed on-device residency, avoiding slow software-managed relocation while retaining many eviction victims locally. Compared with UVMMU, GZswap addresses a different bottleneck: although UVMMU reduces fault latency through hardware migration, evicted pages still move to far memory. GZswap instead restores many reused pages directly from the Zpool through local decompression, reducing repeated off-package transfers under increasing memory pressure.

The overall benefit grows as memory pressure intensifies. Relative to TBNE, GZswap improves performance by 32% at R90, 44% at R75, 56% at R50, and 59% at R35. These gains indicate that compressed on-device residency becomes particularly effective in the working-set-overflow regime for these workloads, where repeated far-memory relocation increasingly dominates execution time.

The Zpool compression ratios further illustrate how workload compressibility affects effective capacity. For example, RO-RA achieves a compression ratio of 9.558× and maintains 92.3% normalized IPC at R35, while all baselines fall below 12%. Its high compressibility allows more eviction victims to remain in the Zpool, reducing performance degradation under severe memory pressure.

In contrast, PB-SGEM exhibits poor compressibility (1.132×), limiting the effective capacity provided by the Zpool. Its compressed page-groups occupy more zpages, increasing Zpool pressure and causing more frequent reclamation to Remote-UM. Consequently, compressed residency provides substantially less benefit for PB-SGEM than for highly compressible workloads such as RO-RA.

The ratios in Figure 9a reflect effective Zpool compression after packing for page-groups stored during simulated execution, rather than the full memory snapshots analyzed in Section 4.1. They therefore capture the compressibility of actual zswap-out victims under runtime memory behavior, including the effect of page packing.

Compression ratio alone, however, does not determine performance. Higher compressibility allows more evicted page-groups to remain in the Zpool, increasing the opportunity for local restoration. This benefit is realized only when a page is reused before its compressed page-group is reclaimed to Remote-UM; otherwise, subsequent access still requires off-package migration. Thus, GZswap performance depends jointly on compressibility, reuse behavior, and the rate of page turnover under memory pressure. These factors also determine how effectively GZswap reduces repeated off-package traffic, which we examine next.

6.3 I/O Traffic

Figure 9b reports I/O traffic, normalized to TBNE at R100. The software-managed baselines, 2MBE and TBNE, suffer from repeated relocations under high memory pressure because reused pages must re-enter the device tier through the far-memory path. Such repeated swap-out and swap-in operations amplify off-package traffic as the

same data may cross the interconnect multiple times during execution. UVMMU reduces some redundant transfers through hardware-based 64 KB page-group migration, but its pseudo-random victim selection can still evict pages that are reused soon after, leading to repeated migrations under oversubscription.

GZswap lowers this overhead more effectively. Although it uses the same pseudo-random Local-UM victim selection as UVMMU, the Zpool acts as an on-device intermediate layer between local and remote memory for recently evicted data. This common victim-selection policy isolates the benefit of compressed residency from differences in eviction policy. Pages that would otherwise be fetched again from far memory can often be restored directly from compressed on-device storage. As memory pressure increases, this local restoration becomes increasingly valuable because repeated reuse would otherwise generate additional off-package transfers. As a result, GZswap reduces I/O traffic by 36% at R90, 46% at R75, 51% at R50, and 55% at R35 relative to TBNE. These results show that GZswap reduces I/O amplification by converting repeated off-package relocations into local compressed-memory accesses.

6.4 Energy Consumption

For energy analysis, we extend AccelWattch [37] to model on-device GPU components, including the compression/decompression engine (CDE), as well as host DRAM and the CPU–GPU interconnect. For host DRAM, we derive module-level dynamic energy in pJ/bit from Micron IDD specification sheets using the difference between the active and idle currents, i.e., IDD4 and IDD0 [52]. For HBM and the CPU–GPU interconnect, we use previously validated energy models [32, 64]. Together, these components capture both the additional on-device activity introduced by GZswap and the off-package activity it eliminates across the modeled system.

For the CDE, we use the compression and decompression energy costs reported for BPC [42] and Deflate [65] and incorporate them into AccelWattch. Because GZswap evaluates BPC and Deflate in parallel during compression, each compression operation accounts for the energy consumed by both engines. During decompression, the compression algorithm is already recorded in the ZDE, so only the selected decompression engine is charged. We also account for the additional HBM accesses incurred by compression and decompression. Thus, our model explicitly includes the compute and memory-energy costs introduced by compressed residency rather than treating compression as a free operation.

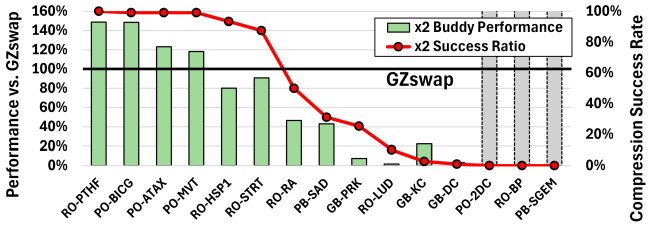
Figure 9c shows the energy breakdown at R50, where only half of the application footprint fits in device memory. GZswap reduces both execution time and off-package traffic, lowering multiple components of system energy simultaneously. First, higher IPC shortens runtime and therefore reduces time-proportional energy. Second, replacing PCIe and host-memory traffic with local HBM accesses and on-device compression/decompression substantially reduces I/O-related energy. Although GZswap incurs additional CDE and HBM activity, these local costs are smaller than the energy avoided through shorter execution and reduced off-package data movement. Compared with TBNE at R50, GZswap reduces static energy by 37% and I/O energy by 55%, resulting in a total energy reduction of 29%.

Relative to TBNE, similar benefits appear at other memory-pressure levels: total energy decreases by 14% at R90, 20% at R75, and 24% at R35. Overall, despite the energy consumed by both compression engines and the additional HBM activity, the reduction in execution time and off-package traffic more than offsets these overheads, yielding total energy savings of 14–29% across the evaluated memory-pressure levels.

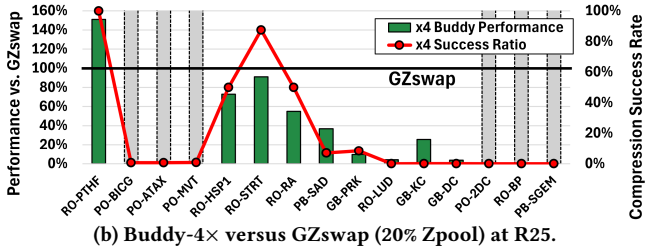
6.5 Comparison with Buddy Compression

We compare GZswap with Buddy Compression [20] to position GZswap with respect to prior GPU memory compression. Buddy Compression achieves high efficiency when nearly all cache lines compress within the target size, minimizing overflow data in host memory. Achieving this, however, requires a consistent compression ratio across blocks and fine-tuning the target compression ratio from prior knowledge; without such tuning, performance degrades in the general case. In contrast, GZswap performs compression on the page-movement path, enabling stronger page-granularity compression and variable-size storage in the on-device Zpool without requiring a fixed compression target per cacheline.

We evaluate fixed 2× and 4× Buddy at R50 and R25, respectively, using the same GPU-memory budget and off-package link as GZswap. To expose target sensitivity, per-allocation target selection is disabled. For Buddy, the *compression success rate* denotes the fraction of 128 B entries that satisfy the target and remain GPU-local. A higher success rate thus reduces remote accesses to buddy memory.



(a) Buddy-2× versus GZswap (20% Zpool) at R50.



(b) Buddy-4× versus GZswap (20% Zpool) at R25.

Figure 10: Buddy performance normalized to GZswap with a 20% Zpool, together with compression success rates. Gray bars denote runs that did not complete in time.

Figure 10a shows that RO-PTHF, PO-BICG, PO-ATAX, and PO-MVT achieve 99–100% success at the 2× target. Thus, nearly all entries remain GPU-local, allowing normal GPU-memory accesses and enabling Buddy to match or outperform GZswap. In other workloads, however, even a small increase in buddy-memory accesses can substantially degrade performance. When target-missing cache lines are frequently reused, their overflow sectors repeatedly traverse the off-package link, turning a small fraction of compression failures into many fine-grained remote accesses. This effect becomes more pronounced in highly memory-intensive workloads

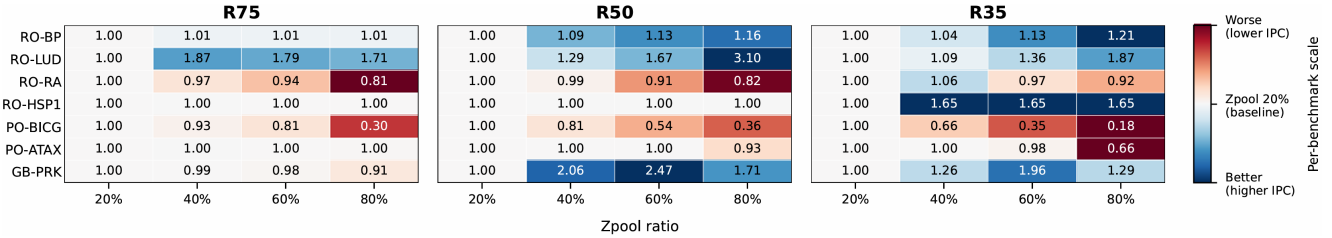


Figure 11: Zpool-size sensitivity under different memory-pressure levels. Each cell shows IPC normalized to the default 20% Zpool configuration for the same workload and memory-pressure level; higher is better.

and over a high-latency PCIe link, where these accesses resemble long-latency, fine-grained zero-copy accesses.

Figure 10b makes this limitation more apparent. Several workloads that satisfy the 2× target no longer meet the more aggressive 4× target: PO-BICG, PO-ATAX, and PO-MVT drop from approximately 99% success at 2× to below 1% at 4×. At cacheline granularity, the lightweight compression used by Buddy exposes less redundancy, making aggressive targets harder to satisfy. Consequently, these workloads incur substantially more buddy-memory accesses and sharply lower performance relative to GZswap.

Overall, Buddy is effective when a moderate fixed target keeps most entries GPU-local, but aggressive targets can sharply increase failures and expose the fine-grained remote path. GZswap instead uses stronger page-granularity compression with variable-size Zpool storage, avoiding fixed-target failures while reducing reliance on costly interconnects such as NVLink [82].

6.6 Zpool Sensitivity

We further evaluate the sensitivity of GZswap to Zpool capacity across memory-pressure levels using Zpool ratios of 20/40/60/80%. Figure 11 reports IPC normalized to the default 20% Zpool configuration at R75, R50, and R35; higher is better. For readability, we omit benchmarks whose IPC remains within ±10% of the default-Zpool baseline across all non-default Zpool ratios.

Because the Zpool is carved out of the same device-memory budget, increasing its size expands compressed capacity while reducing directly accessible Local-UM capacity. The preferred Zpool ratio therefore depends on both workload behavior and memory pressure during execution, reflecting a trade-off between compressed residency and immediately accessible capacity.

For example, RO-LUD performs best with a 40% Zpool at R75, reaching 1.87× the IPC of the default configuration, whereas larger Zpools become more beneficial at R50 and R35, where the 80% configuration reaches 3.10× and 1.87×, respectively. In contrast, PO-BICG consistently performs best with the default 20% Zpool, indicating that reduced Local-UM capacity can outweigh the benefit of additional compressed residency.

GB-PRK further illustrates that the optimum can lie between these extremes under varying pressure for this workload. At R50, a 60% Zpool achieves 2.47× the IPC of the default configuration, outperforming both the 40% (2.06×) and 80% (1.71×) configurations. A similar non-monotonic trend appears at R35, where the 60% Zpool achieves 1.96× compared with 1.26× and 1.29× at 40% and 80%, respectively. These results show that maximizing Zpool capacity alone does not necessarily maximize performance.

Overall, a larger Zpool is not universally beneficial, and the preferred partition depends on workload characteristics and memory pressure. Characterizing working-set dynamics and dynamically adjusting the Zpool ratio at runtime are left for future work.

7 Conclusion

This paper presented GZswap, a hardware-managed on-device compressed swap mechanism for GPU memory oversubscription. We showed that software-coherent Unified Memory suffers from repeated off-package relocation when the active working set exceeds device-memory capacity. GZswap addresses this problem by retaining selected eviction victims in compressed form inside GPU memory and restoring them through GPU-side fault handling.

By converting off-package migrations into on-device compression, decompression, and Zpool accesses, GZswap expands effective near-memory capacity and reduces relocation traffic. Our evaluation shows that GZswap improves IPC by 32–59%, reduces relocation I/O traffic by 36–55%, and lowers total energy by up to 29%. These results demonstrate that on-device compressed swap is a practical complement to existing GPU memory-expansion mechanisms under memory oversubscription.

Acknowledgments

This work was supported in part by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grants funded by the Korea Government (MSIT) under Grants RS-2023-00228970 (Development of Flexible SW-HW Conjunctive Solution for On-edge Self-supervised Learning, 20%), RS-2026-25519569 (Development of an Operating Interface and Platform for LPDDR6-PIM, 20%), and RS-2026-25520910 (Development of LPDDR6 Interface, Low-power and High-speed Memory Network, 20%); and in part by Samsung Electronics Company, Ltd. under Grant IO251219-14883-01 (40%).

References

[1] Bulent Abali, Bart Blaner, John Reilly, Matthias Klein, Ashutosh Mishra, Craig B. Agricola, Bedri Sendir, Alper Buyuktosunoglu, Christian Jacobi, William J. Starke, Haren Myneni, and Charlie Wang. 2020. Data compression accelerator on IBM POWER9 and z15 processors. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 1–14. doi:10.1109/ISCA45697.2020.00012

[2] Alaa R. Alameldeen and David A. Wood. 2004. *Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches.* Technical Report CS-TR-2004-1500. Department of Computer Sciences, University of Wisconsin–Madison.

[3] 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

[4] Tyler Allen and Rong Ge. 2021. Demystifying GPU UVM Cost with Deep Runtime and Workload Analysis. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 141–150. doi:10.1109/IPDPS49936.2021.00023

[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] Alexandra Angerd, Angelos Arelakis, Vasilis Spiliopoulos, Erik Sintorn, and Per Stenström. 2022. GBDI: Going Beyond Base-Delta-Immediate Compression with Global Bases. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1115–1127. doi:10.1109/HPCA53966.2022.00085

[7] Angelos Arelakis, Fredrik Dahlgren, and Per Stenstrom. 2015. HyComp: a hybrid cache compression method for selection of data-type-specific compression methods. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 38–49. doi:10.1145/2830772.2830823

[8] Angelos Arelakis and Per Stenstrom. 2014. SC2: a statistical compression cache scheme. *SIGARCH Comput. Archit. News* 42, 3 (June 2014), 145–156. doi:10.1145/2678373.2665696

[9] Akhil Arunkumar, Shin-Ying Lee, Vignesh Soundararajan, and Carole-Jean Wu. 2018. LATTE-CC: Latency Tolerance Aware Adaptive Cache Compression Management for Energy Efficient GPUs. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 221–234. doi:10.1109/HPCA.2018.00028

[10] 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

[11] Joshua Bakita and James H. Anderson. 2022. Enabling GPU Memory Oversubscription via Transparent Paging to an NVMe SSD. In *2022 IEEE Real-Time Systems Symposium (RTSS)*. 370–382. doi:10.1109/RTSS55097.2022.00039

[12] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Trans. Archit. Code Optim.* 14, 2, Article 14 (June 2017), 25 pages. doi:10.1145/3085572

[13] 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

[14] Shuai Che, Bradford M. Beckmann, Steven K. Reinhardt, and Kevin Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. 185–195. doi:10.1109/IISWC.2013.6704684

[15] 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

[16] Xi Chen, Lei Yang, Robert P. Dick, Li Shang, and Haris Lekatsas. 2010. C-pack: a high-performance microprocessor cache compression algorithm. *IEEE Trans. Very Large Scale Integr. Syst.* 18, 8 (Aug. 2010), 1196–1208. doi:10.1109/TVLSI.2009.2020989

[17] Chirayu Garg and Nikolay Sakharnykh. 2023. Improving GPU Memory Oversubscription Performance. https://developer.nvidia.com/blog/improving-gpu-memory-oversubscription-performance/ Accessed: 2026-04-29.

[18] Sangjin Choi, Taeksoo Kim, Jinwoo Jeong, Rachata Ausavarungnirun, Myeongjae Jeon, Youngjin Kwon, and Jeongseob Ahn. 2022. Memory Harvesting in Multi-GPU Systems with Hierarchical Unified Virtual Memory. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 625–638. https://www.usenix.org/conference/atc22/presentation/choi-sangjin

[19] Esha Choukse, Mattan Erez, and Alaa R. Alameldeen. 2018. Compresso: Pragmatic Main Memory Compression. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 546–558. doi:10.1109/MICRO.2018.00051

[20] Esha Choukse, Michael B. Sullivan, Mike O'Connor, Mattan Erez, Jeff Pool, David Nellans, and Stephen W. Keckler. 2020. Buddy Compression: Enabling Larger Memory for Deep Learning and HPC Workloads on GPUs. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 926–939. doi:10.1109/ISCA45697.2020.00080

[21] Peter J. Denning. 2021. Working Set Analytics. *ACM Comput. Surv.* 53, 6, Article 113 (Feb. 2021), 36 pages. doi:10.1145/3399709

[22] L. Peter Deutsch. 1996. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951. doi:10.17487/RFC1951

[23] Debashis Ganguly, Rami Melhem, and Jun Yang. 2021. An Adaptive Framework for Oversubscription Management in CPU-GPU Unified Memory. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1212–1217. doi:10.23919/DATE51398.2021.9473982

[24] 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

[25] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. 2020. Adaptive Page Migration for Irregular Data-intensive Applications under GPU Memory Oversubscription. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 451–461. doi:10.1109/IPDPS47924.2020.00054

[26] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. 2024. AI and Memory Wall. *IEEE Micro* 44, 3 (May 2024), 33–39. doi:10.1109/MM.2024.3373763

[27] Google. [n. d.]. Snappy: A Fast Compressor/Decompressor. https://google.github.io/snappy/. Accessed: 2026-04-30.

[28] Donghyun Gouk, Seungkwan Kang, Hanyeoreum Bae, Eojin Ryu, Sangwon Lee, Dongpyung Kim, Junhyeok Jang, and Myoungsoo Jung. 2024. Breaking Barriers: Expanding GPU Memory with Sub-Two Digit Nanosecond Latency CXL Controller. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems* (Santa Clara, CA, USA) *(HotStorage '24)*. Association for Computing Machinery, New York, NY, USA, 108–115. doi:10.1145/3655038.3665953

[29] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar)*. 1–10. doi:10.1109/InPar.2012.6339595

[30] Yongbin Gu, Wenxuan Wu, Yunfan Li, and Lizhong Chen. 2020. UVMBench: A Comprehensive Benchmark Suite for Researching Unified Virtual Memory in GPUs. *CoRR* abs/2007.09822 (2020). arXiv:2007.09822 https://arxiv.org/abs/2007.09822

[31] Apoorv Gupta, Aman Bansal, and Vidhi Khanduja. 2017. Modern lossless compression techniques: Review, comparison and analysis. In *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. 1–8. doi:10.1109/ICECCT.2017.8117850

[32] Jeongmin Hong, Sungjun Cho, Geonwoo Park, Wonhyuk Yang, Young-Ho Gong, and Gwangsun Kim. 2024. Bandwidth-Effective DRAM Cache for GPUs with Storage-Class Memory. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 139–155. doi:10.1109/HPCA57654.2024.00021

[33] Seokin Hong, Prashant Jayaprakash Nair, Bulent Abali, Alper Buyuktosunoglu, Kyu-Hyoun Kim, and Michael Healy. 2018. Attaché: Towards Ideal Memory Compression by Mitigating Metadata Bandwidth Overheads. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 326–338. doi:10.1109/MICRO.2018.00034

[34] Intel Corporation. 2025. *Memory-Tiering with Intel® In-Memory Analytics Accelerator*. https://www.intel.com/content/www/us/en/content-details/865249/memory-tiering-with-intel-in-memory-analytics-accelerator.html

[35] Houxiang Ji, Mark Mansi, Yan Sun, Yifan Yuan, Jinghan Huang, Reese Kuper, Michael M. Swift, and Nam Sung Kim. 2023. STYX: Exploiting SmartNIC Capability to Reduce Datacenter Memory Tax. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 619–633. https://www.usenix.org/conference/atc23/presentation/ji

[36] Hoyong Jin, Donghun Jeong, Taewon Park, Jong Hwan Ko, and Jungrae Kim. 2022. Multi-Prediction Compression: An Efficient and Scalable Memory Compression Framework for GP-GPU. *IEEE Computer Architecture Letters* 21, 2 (2022), 37–40. doi:10.1109/LCA.2022.3177419

[37] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G. Rogers, Tor M. Aamodt, and Nikos Hardavellas. 2021. AccelWattch: A Power Modeling Framework for Modern GPUs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 738–753. doi:10.1145/3466752.3480063

[38] Aajna Karki, Chethan Palangotu Keshava, Spoorthi Mysore Shivakumar, Joshua Skow, Goutam Madhukeshwar Hegde, and Hyeran Jeon. 2019. Detailed Characterization of Deep Neural Networks on GPUs and FPGAs. In *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs* (Providence, RI, USA) *(GPGPU '19)*. Association for Computing Machinery, New York, NY, USA, 12–21. doi:10.1145/3300053.3319418

[39] 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

[40] 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

[41] Jongseok Kim, Cheolgi Kim, and Euiseong Seo. 2019. *ezswap* : Enhanced Compressed Swap Scheme for Mobile Devices. *IEEE Access* 7 (2019), 139678–139691. doi:10.1109/ACCESS.2019.2942362

[42] Jungrae Kim, Michael Sullivan, Esha Choukse, and Mattan Erez. 2016. Bit-Plane Compression: Transforming Data for Better Compression in Many-Core Architectures. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 329–340. doi:10.1109/ISCA.2016.37

[43] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 15, 1 (2016), 45–49. doi:10.1109/LCA.2015.2414456

[44] Sandeep Kumar, Aravinda Prasad, and Sreenivas Subramoney. 2026. TierScape: Harnessing Multiple Compressed Tiers to Tame Server Memory TCO. In *Proceedings of the 21st European Conference on Computer Systems* (McEwan Hall/The University of Edinburgh, Edinburgh, Scotland UK) *(EUROSYS '26)*. Association for Computing Machinery, New York, NY, USA, 247–262. doi:10.1145/3767295.3769321

[45] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. 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, 317–330. doi:10.1145/3297858.3304053

[46] Sohan Lal, Bogaraju Sharatchandra Varma, and Ben Juurlink. 2022. A Quantitative Study of Locality in GPU Caches for Memory-Divergent Workloads. *International Journal of Parallel Programming* 50, 2 (01 Apr 2022), 189–216. https://doi.org/10.1007/s10766-022-00729-2

[47] Jonas Latt, Christophe Guy Coreixas, Gonzalo Brito, and Jeff Larkin. 2022. Multi-GPU Programming with Standard Parallel C++, Part 1. https://developer.nvidia.com/blog/multi-gpu-programming-with-standard-parallel-c-part-1/. Accessed: 2026-04-30.

[48] Gyusun Lee, Jinyong Park, Jae W. Lee, and Jinkyu Jeong. 2025. Transparent Smartphone Memory Expansion. In *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing* (Catania International Airport, Catania, Italy) *(SAC '25)*. Association for Computing Machinery, New York, NY, USA, 1325–1334. doi:10.1145/3672608.3707936

[49] 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

[50] Mao Lin, Yuan Feng, Guilherme Cox, and Hyeran Jeon. 2025. Forest: Access-aware GPU UVM Management. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA '25)*. Association for Computing Machinery, New York, NY, USA, 137–152. doi:10.1145/3695053.3731047

[51] LZ4 Project. 2026. LZ4: Extremely Fast Compression Algorithm. https://lz4.org/. Accessed: 2026-04-30.

[52] Micron Technology, Inc. 2023. *32GB (x80, ECC, SR) 288-Pin DDR5 RDIMM Features and IDD Specifications (MTC20F1045S1RC – 32GB 16Gb Die Revision A)*. https://www.micron.com/products/dram-modules/rdimm/part-catalog/mtc20f1045s1rc-32gb

[53] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. 2015. GraphBIG: understanding graph computing in the context of industrial solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 69, 12 pages. doi:10.1145/2807591.2807626

[54] Nurlan Nazaraliyev, Elaheh Sadredini, and Nael Abu-Ghazaleh. 2025. DREAM: Device-Driven Efficient Access to Virtual Memory. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS '25)*. Association for Computing Machinery, New York, NY, USA, 1190–1205. doi:10.1145/3721145.3725748

[55] Abdun Nihaal and Madhu Mutyam. 2024. Selective Memory Compression for GPU Memory Oversubscription Management. In *Proceedings of the 53rd International Conference on Parallel Processing* (Gotland, Sweden) *(ICPP '24)*. Association for Computing Machinery, New York, NY, USA, 189–198. doi:10.1145/3673038.3673058

[56] NVIDIA Corporation. 2017. GP100 MMU Format Specification. https://nvidia.github.io/open-gpu-doc/pascal/gp100-mmu-format.pdf.

[57] NVIDIA Corporation. 2020. *NVIDIA A100 Tensor Core GPU Architecture*. Technical Report. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[58] NVIDIA Corporation. 2025. *NVIDIA Nsight™ Compute*. https://developer.nvidia.com/nsight-compute

[59] NVIDIA Corporation. 2025. *NVIDIA Nsight™ Systems*. https://developer.nvidia.com/nsight-systems

[60] NVIDIA Corporation. 2026. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html Version 13.2, March 2026.

[61] NVIDIA Corporation. 2026. NVIDIA Blackwell Architecture. https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/. Accessed: 2026-04-29.

[62] NVIDIA Corporation. 2026. NVIDIA GH200 Grace Hopper Superchip. https://www.nvidia.com/en-us/data-center/grace-hopper-superchip/. Accessed: 2026-04-29.

[63] NVIDIA Corporation. 2026. NVIDIA nvCOMP. https://developer.nvidia.com/nvcomp. Accessed: 2026-04-29.

[64] Mike O'Connor, Niladrish Chatterjee, Donghyuk Lee, John Wilson, Aditya Agrawal, Stephen W. Keckler, and William J. Dally. 2017. Fine-grained DRAM: energy-efficient DRAM for extreme bandwidth systems. 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, 41–54. doi:10.1145/3123939.3124545

[65] Gagandeep Panwar, Muhammad Laghari, David Bears, Yuqing Liu, Chandler Jearls, Esha Choukse, Kirk W. Cameron, Ali R. Butt, and Xun Jian. 2022. Translation-Optimized Memory Compression for Capacity. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture* (Chicago, Illinois, USA) *(MICRO '22)*. IEEE Press, 992–1011. doi:10.1109/MICRO56248.2022.00073

[66] Jihun Park, Donghun Jeong, and Jungrae Kim. 2023. UVMMU: Hardware-Offloaded Page Migration for Heterogeneous Computing. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1–6. doi:10.23919/DATE56975.2023.10137307

[67] Junhyeok Park, Osang Kwon, Yongho Lee, Seongwook Kim, Gwangeun Byeon, Jihun Yoon, Prashant J. Nair, and Seokin Hong. 2024. A Case for Speculative Address Translation with Rapid Validation for GPUs. In *Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture* (Austin, TX, USA) *(MICRO '24)*. IEEE Press, 278–292. doi:10.1109/MICRO61859.2024.00029

[68] Neel Patel, Amin Mamandipoor, Derrick Quinn, and Mohammad Alian. 2023. XFM: Accelerated Software-Defined Far Memory. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 769–783. doi:10.1145/3613424.3623776

[69] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2012. Base-delta-immediate compression: practical data compression for on-chip caches *(PACT '12)*. Association for Computing Machinery, New York, NY, USA, 377–388. doi:10.1145/2370816.2370870

[70] James L. Peterson. 1990. Discussion of Inverted Page Tables. *RT/RIOS Architecture* (1990). https://jklp.org/profession/papers/ipt/ipt.html

[71] 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

[72] 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

[73] Nikolay Sakharnykh. 2016. Beyond GPU Memory Limits: Unified Memory and the Pascal Architecture. https://developer.nvidia.com/blog/beyond-gpu-memory-limits-unified-memory-pascal/

[74] Chuanming Shao, Jinyang Guo, Pengyu Wang, Jing Wang, Chao Li, and Minyi Guo. 2022. Oversubscribing GPU Unified Virtual Memory: Implications and Suggestions. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering* (Beijing, China) *(ICPE '22)*. Association for Computing Machinery, New York, NY, USA, 67–75. doi:10.1145/3489525.3511691

[75] John A. Stratton, Christopher I. Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng D. Liu, and Wen-mei W. Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Technical Report IMPACT-12-01. University of Illinois at Urbana-Champaign.

[76] TechPowerUp. 2020. NVIDIA GeForce RTX 3090 Specs. https://www.techpowerup.com/gpu-specs/geforce-rtx-3090.c3622.

[77] TechPowerUp. 2026. AMD EPYC 7313 Specs. https://www.techpowerup.com/cpu-specs/epyc-7313.c2389. Accessed: 2026-04-29.

[78] The Linux Kernel Development Community. 2019. *zswap: Compressed Cache for Swap Pages*. https://www.kernel.org/doc/html/v5.0/vm/zswap.html

[79] The Linux Kernel Development Community. 2025. *perf: Linux Profiling with Performance Counters*. https://perf.wiki.kernel.org/

[80] 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

[81] Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. 2023. YOLOv7: Trainable Bag-of-Freebies Sets New State-of-the-Art for Real-Time Object Detectors. In *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 7464–7475. doi:10.1109/CVPR52729.2023.00721

[82] Ying Wei, Yi Chieh Huang, Haiming Tang, Nithya Sankaran, Ish Chadha, Dai Dai, Olakanmi Oluwole, Vishnu Balan, and Edward Lee. 2023. 9.3 NVLink-C2C: A Coherent Off Package Chip-to-Chip Interconnect with 40Gbps/pin Single-ended Signaling. In *2023 IEEE International Solid-State Circuits Conference (ISSCC)*. 160–162. doi:10.1109/ISSCC42615.2023.10067395

[83] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 609–621. doi:10.1145/3503222.3507731

[84] Qirong Xia, Houxiang Ji, Yang Zhou, and Nam Sung Kim. 2025. Hardware-Accelerated Kernel-Space Memory Compression Using Intel QAT. *IEEE Computer Architecture Letters* 24, 1 (2025), 57–60. doi:10.1109/LCA.2025.3534831

[85] Qi Yu, Bruce Childers, Libo Huang, Cheng Qian, Hui Guo, and Zhiying Wang. 2020. Coordinated Page Prefetch and Eviction for Memory Oversubscription Management in GPUs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 472–482. doi:10.1109/IPDPS47924.2020.00056

[86] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W. Keckler. 2016. Towards high performance paged memory for GPUs. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 345–357. doi:10.1109/HPCA.2016.7446077