ESA: Improving GPU Utilization with Elastic Isolation for ML Inference Services

Taeklim Kim
The University of Texas at Austin
Austin, Texas, USA
tlkim@cs.utexas.edu

Saurabh Agarwal
The University of Texas at Austin
Austin, Texas, USA
saurabh.agarwal@utexas.edu

Rachata Ausavarungnirun
MangoBoost Inc.
Skokie, Illinois, USA
r.ausavarungnirun@gmail.com

Jayneel Gandhi
Meta
Menlo Park, California, USA
jayneel@meta.com

Christopher J. Rossbach
The University of Texas at Austin and
Microsoft
Austin, Texas, USA
rossbach@cs.utexas.edu

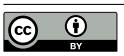
Abstract

GPU compute density continues to increase, exposing plentiful resources that often remain under-utilized. ML inference, for example, often serves small batch requests and over-provisions resources to meet strict peak-load latency deadlines, leading to significant under-utilization. Sharing GPUs across applications could improve utilization, but current techniques are inadequate. Elastic sharing systems, such as NVIDIA MPS, improve utilization, but allow interference to degrade application performance. In contrast, strict hardware partitioning, such as NVIDIA MIG, provides strong isolation, but its inelasticity results in under-utilization and suboptimal performance. Users need a way to better utilize GPUs without compromising performance for critical ML services.

ESA is a novel GPU design for priority-aware elastic sharing *with* performance isolation. Instead of relying on static partitioning or unrestricted sharing, *ESA* detects performance-critical interference using performance monitors and dynamically composes isolation mechanisms across the interconnect, cache, and memory hierarchy. By activating isolation *only when needed*, *ESA* allows best-effort applications to use spare resources while preserving predictable performance for critical services. *ESA* combines four mechanisms with modest hardware costs: interconnect injection throttling, virtual channel partitioning, way-based cache partitioning, and memory request prioritization. Because no single mechanism is effective across all contention scenarios, *ESA* selects mechanisms based on user-defined priorities and observed bottlenecks, enabling both QoS protection and fair sharing. Our evaluation shows that *ESA* enables latency-sensitive applications, such as LLM serving, to share a GPU with co-running applications while incurring near-zero performance impact from contention. In fair-sharing scenarios, *ESA* improves multi-application throughput by up to 1.67× over MPS and 1.49× over MIG.

CCS Concepts

• **Computer systems organization → Parallel architectures**.



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/10.1145/3838684.3846854

Keywords

GPU architecture, GPU sharing, ML inference

ACM Reference Format:
Taeklim Kim, Saurabh Agarwal, Rachata Ausavarungnirun, Jayneel Gandhi, and Christopher J. Rossbach. 2026. *ESA*: Improving GPU Utilization with Elastic Isolation for ML Inference Services. 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/10.1145/3838684.3846854

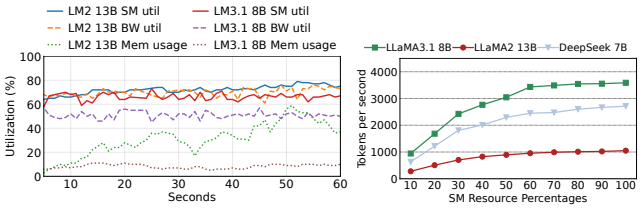
1 Introduction

Increasing GPU compute density has driven advances in ML, computer vision, and generative AI. Modern GPUs provide abundant resources, including hundreds of streaming multiprocessors (SMs), thousands of concurrent threads, and large memory capacity. However, in practice, many applications fail to fully utilize these resources. Recent studies [42, 111, 112, 116] report significant GPU under-utilization in production environments: exclusive GPU allocation at Microsoft leads to mean utilization of only about 50% [42], while Alibaba reports utilization as low as 10% [111, 116].

Low utilization is especially common in ML inference serving [20, 27, 28, 34, 35, 70, 92, 111]. Inference typically uses small batches to meet strict service-level objectives (SLOs), and cloud providers often over-provision GPUs for peak load [20, 27, 28, 34]. Figure 1a shows that two LLM serving workloads on an NVIDIA H100 GPU leave substantial resources under-utilized.[1] For LLaMA3.1 8B, average SM utilization is only 64%, and average memory bandwidth utilization is only 51%. These results show that latency-sensitive LLM serving can leave significant GPU resources idle, creating an opportunity to use spare resources for other workloads.

Moreover, the bottleneck resource varies across kernels and workloads, ranging from compute to memory capacity and bandwidth [58, 78, 93, 131, 132]. LLM inference, for example, has distinct phases: prompt-processing kernels are primarily compute-bound due to parallel processing of the full input sequence, whereas decoding kernels are memory-bandwidth-bound because of frequent access to attention key-value tensors. LLM schedulers that prioritize decode phases, such as chunked-prefill [1], further increase the fraction of mixed batches dominated by decode-side memory traffic. Figure 1b shows that LLM inference on an NVIDIA A100 with

[1] We revisit the methodology behind these experiments in §3.



(a) GPU utilization during LLM inference **(b)** LLM throughput under varying SM allocations

Figure 1: Evidence of GPU resource under-utilization.

MPS [74] gains little performance beyond 60% SM usage, suggesting that additional SM resources provide limited benefit.

GPU sharing through spatial partitioning can improve utilization by co-locating multiple ML tasks on a single GPU, but practical sharing must preserve performance for latency-critical inference tasks. Despite a wealth of prior work [3, 22, 29, 30, 36, 47, 52, 55, 97, 121] and production systems [23, 29, 36, 74], GPU sharing remains challenging due to the fundamental tradeoffs between elasticity and performance isolation. *Elasticity* is important for improving utilization and throughput by dynamically reallocating resources in response to changing demand, but fine-grained sharing introduces contention. *Performance isolation* ensures predictable performance, yet rigid partitioning leads to resource fragmentation and under-utilization. Balancing these competing goals is difficult: lack of elasticity wastes resources, while lack of isolation exposes applications to increased tail latency and missed QoS targets.

Current GPU sharing techniques force users to choose between isolation and utilization. NVIDIA MIG [23] statically partitions memory, on-chip interconnect, and SMs into virtual GPUs. While this provides strong isolation, it hurts ML serving performance by limiting access to memory bandwidth and fragmenting resources, causing under-utilization. Conversely, elastic sharing systems [7, 8, 29, 30, 80, 97, 121], such as NVIDIA MPS [74], enable fine-grained resource sharing, improving utilization and system-wide throughput, but introduce performance loss from contention for shared microarchitectural resources (§3).

Prior work has proposed GPU sharing via software or hardware, but none provides both elasticity and performance isolation (Table 1). Software-only approaches [61, 92, 99, 123, 124, 127] improve utilization by sharing GPUs between ML inference and best-effort jobs, but they rely on coarse-grained partitioning and lack mechanisms to manage microarchitectural contention. Some also require kernel rewrites for persistent-thread co-scheduling or profiling to capture performance behavior. Hardware-based approaches [38, 69, 81, 125] typically isolate individual components such as cache or memory bandwidth, but do not provide a comprehensive solution that applies the right combination of isolation techniques across all shared resources at the appropriate time.

We present *ESA* (**E**lastic **S**hared **A**ccelerators), a novel design that provides both elasticity and performance isolation for prioritized applications, such as ML inference, while supporting best-effort background jobs. *ESA*'s key idea is to let applications *freely use resources until contention arises*, at which point it activates a tailored mix of mediation techniques. Based on user-defined priorities, *ESA* dynamically deploys interconnect injection throttling (*IT*), virtual channel partitioning (*VCP*), way-based cache partitioning (*CP*), and memory request prioritization (*MRP*).

We show that different applications and mixes respond differently to each isolation technique. No single technique works best in all cases. *ESA*'s key novelty is a set of lightweight techniques that transparently observe application behavior to selectively apply mediation. Our GPGPU-Sim prototype shows that *ESA* provides performance isolation comparable to, and often better than, MIG while avoiding fragmentation and under-utilization. *ESA* also allows users to specify task priorities, enabling highest-priority tasks such as inference to run at near-uncontended performance while sharing the GPU with best-effort jobs. We make the following contributions:

- We characterize the performance isolation challenge that arises when LLM inference and other applications share GPU resources, including the on-chip interconnect and memory subsystems.
- We present a comprehensive analysis of the performance isolation design space, showing that mediated sharing can prioritize latency-critical ML inference to prevent performance degradation, while also supporting fairness among multiple tenants with comparable isolation needs.
- We present novel techniques for mediating shared GPU resources that dynamically apply tailored combinations of interconnect injection throttling, virtual channel partitioning, way-based cache partitioning, and memory request prioritization.
- We show that *ESA* maintains near-uncontended performance for latency-critical applications co-located with best-effort workloads and improves throughput by up to 1.67× in fair-sharing scenarios.

2 Background

GPU sharing. GPUs support virtual memory and memory protection [7, 8, 79, 80], allowing data from different protection domains to safely reside in GPU physical memory. Software sharing systems, such as NVIDIA MPS, enable concurrent execution by aggregating GPU interactions from multiple applications into a single context that mediates device access. Hardware-based partitioning provides stronger isolation between concurrent GPU kernels. NVIDIA MIG [22, 23], for example, partitions a GPU into a small number of fixed-size instances that appear as separate GPUs to applications. On the A100/H100 GPU, MIG supports up to seven partitions using the smallest partition profile. These partitions can be provisioned with dedicated resources to reduce interference, and reconfigured dynamically *as long as they are not in use by any process* [23].

Vertical vs. Horizontal Partitioning. We assume a generic spatial sharing design similar to MPS [74] or MIG [22], as shown in Figure 2, where SMs are partitioned among tenants either statically or dynamically. We classify partitioning of the remaining memory hierarchy resources as *vertical* or *horizontal*. Vertical partitioning dedicates slices of interconnect, L2 cache, and memory channels to different tenants, as in MIG. Horizontal partitioning allows tenants access to shared memory hierarchy resources, relying on memory protection for strong *data isolation*. Current horizontal techniques, such as MPS, improve flexibility but allow tenants to freely compete for shared resources.

Method	Implementation	Interconnect Partitioning	Interconnect Throttling	Shared Cache Partitioning	Dynamic Memory BW Partitioning	No Profiler or Kernel Modification Required
MPS [74]	User + Driver	✗	✗	✗	✗	✓
MIG [23]	Hardware	✓	✗	✓	✗	✓
SW approach [92, 99, 123, 124, 127]	User-space	✗	✗	✗	✗	✗
BW reservation [69]	Hardware	✗	✗	✓	✓	✓
FGPU [38]	Driver	✗	✗	✓	✗	✗
SGDRC [125]	User + Driver	✗	✗	✓	✓	✗
***ESA* (Ours)**	Hardware	✓	✓	✓	✓	✓

Table 1: Comparison of GPU sharing solutions across software and hardware approaches.

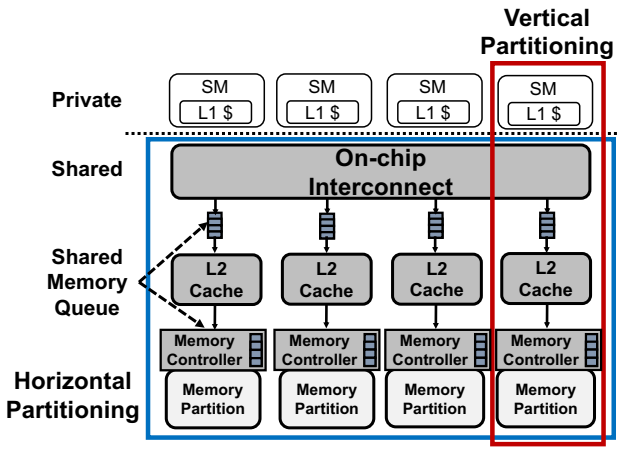


Figure 2: GPU architecture with shared on-chip interconnect and memory subsystems. Vertical partitioning gives each application independent slices of SMs, interconnect, L2 cache, and memory partitions. Horizontal partitioning lets each application use any resources with or without isolation mechanisms.

ML Inference Serving. As ML models become increasingly popular, demand for efficient inference serving systems continues to grow. Unlike training, inference does not always fully utilize both GPU compute and memory resources because small batch sizes are used to meet QoS commitments [20, 34, 35, 70, 92]. Additionally, ML algorithms such as LLMs exhibit variable resource usage patterns. For instance, during the phase of processing the user's prompt, kernels are primarily compute-bound because prompt tokens are processed in parallel to generate the next token. However, during token generation, kernels become memory bandwidth-bound due to heavy accesses to key-value tensors (KV cache), leaving compute resources under-utilized. These characteristics create opportunities for GPU sharing, but sharing must avoid performance interference that can violate inference latency deadlines.

3 Motivation

This section characterizes LLM inference by analyzing its sensitivity to resource availability and contention. We show how co-locating LLM inference with best-effort applications can improve GPU utilization, and identify the performance-isolation challenges that arise in shared resources such as the on-chip interconnect and memory subsystem.

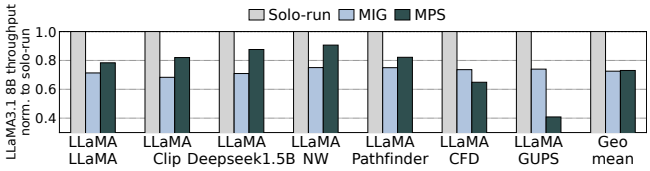


Figure 3: LLM inference throughput while co-running different workloads using MPS and MIG. We allocate sufficient SMs to LLM inference to saturate throughput. LLM throughput varies depending on the tenant applications.

3.1 Under-utilization of ML Inference

3.1.1 Compute Under-utilization of LLM Inference. Figure 1b shows the throughput of vLLM [58] running LLaMA models [102] and DeepSeek [11] on an NVIDIA A100 GPU with 40GB memory, with model weights and KV caches stored in GPU memory. We vary the available compute resources using NVIDIA MPS, adjusting the SM allocation from 10% to 100%. Across all models, performance gains beyond 60% SMs are minimal at best. For example, increasing SM allocation from 60% to 100% improves throughput by less than 4% for `LLaMA3.1 8B` and less than 10% for `LLaMA2 13B` and `DeepSeek 7B`. This limited scaling occurs because token generation is memory-bound, with frequent KV-cache accesses limiting the benefit of additional compute resources [58].

3.1.2 Memory Under-utilization of LLM Inference. Unlike throughput-oriented training, inference serving uses smaller request batches to avoid SLO violations, often leading to low GPU utilization. Figure 1a shows that two LLM serving workloads on an NVIDIA H100 GPU leave substantial resources under-utilized even at the maximum load that satisfies a 100 ms token generation latency target [1, 19, 82, 91, 115]. Using the NVIDIA NVML API [75], we find that average memory bandwidth utilization is only 51% for `LLaMA3.1 8B`. These results show that even under the highest load that satisfies the latency target, substantial GPU resources remain under-utilized; stricter SLO constraints would further reduce utilization.

We also measure KV cache memory usage during inference to quantify under-utilized memory capacity. Figure 1a shows the fraction of reserved KV cache memory that is actually used. Both models use only a portion of their reserved KV-cache memory: `LLaMA3.1 8B` uses only 11%, while `LLaMA2 13B` uses up to 60%. This suggests that operating at batch sizes and loads that satisfy latency SLOs creates opportunities to co-locate other best-effort ML algorithms or HPC workloads [14, 24, 33, 89, 94, 101]. The

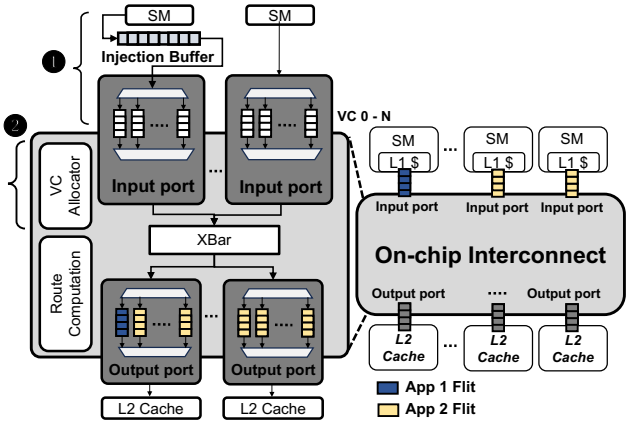


Figure 4: On-chip interconnect and router microarchitecture.

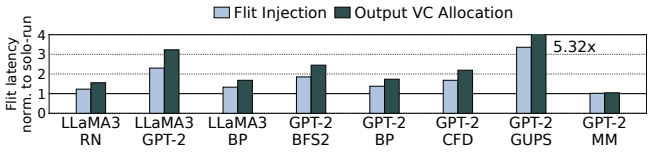


Figure 5: LLaMA3 and GPT-2's (first row) flit latency breakdown on the interconnect when concurrently running with best-effort applications. Each latency is normalized to LLaMA3 and GPT-2's solo-run, showing increases of 65% and 213% due to congestion with tenants.

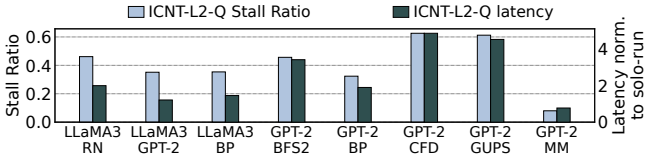


Figure 6: Interference on shared memory queues between interconnect and L2 cache partitions. Both LLaMA3 and GPT-2's memory requests are frequently stalled due to contention from tenant applications, leading to higher queuing delay.

spare memory capacity can accommodate such workloads without necessarily evicting LLM inference's intermediate data.

3.2 Inference Performance on Shared GPUs

While co-locating LLM inference with other applications can improve utilization, existing sharing techniques can cause unacceptable performance loss for LLM inference. We measure LLM throughput while co-running applications from diverse inference workflows, including speculative decoding with a small draft LLM [63], an image encoder for image classification [76], and best-effort HPC workloads [14, 24, 37]. Using the same configuration as Figure 1b, we allocate 56% of SMs to LLM inference and 42% to the co-running applications, corresponding to 4 and 3 GPC (Graphics Processing Cluster) slices when using MIG.

On a real A100 GPU, Figure 3 shows that LLM performance loss depends strongly on which application it is co-located with. Co-running `LLaMA` with memory-bound applications, such as `CFD` or another `LLaMA` model, heavily stresses the shared cache and memory bandwidth, significantly degrading LLM performance. In contrast, co-running with `NW` or a small draft model, `DeepSeek1.5B`, causes minimal degradation because these workloads are compute-bound. Although the performance impact varies across workloads, such degradation is unacceptable for production ML serving, motivating the need to isolate LLM performance by managing contended resources. §7 shows that *ESA* can completely recover the lost performance.

3.3 Interference on On-chip Interconnect

Horizontal partitioning (Figure 2) enables multiple applications to share the on-chip interconnect that connects SMs to L2 cache partitions. Each application privately uses its allocated SMs and L1 caches, while L1 cache misses generate memory fetch requests that traverse the shared interconnect. Figure 4 illustrates key interconnect components, including the network interface and routers. This is a widely adopted design in GPU microarchitecture [44, 57, 60, 128]. Memory requests are split into flits and injected through SM-side input ports. Each input port contains multiple virtual channels (VCs), which buffer flits, while a VC allocator assigns available channels. After switch allocation, flits traverse the crossbar to reach output ports connected to L2 cache partitions. Because any SM can access any L2 cache partition, output port VC allocation can easily become a contention point across co-running applications. For instance, an application that sends a high volume of memory requests can monopolize all available VCs and block other applications from accessing memory.

Recent GPUs use an interconnect with a hierarchical crossbar design, consisting of clusters that contain multiple SMs or memory partitions. Each cluster utilizes either an SM-side router or a memory-side router [128–130]. This two-level router design reduces interconnect complexity and enhances scalability. However, the hierarchical crossbar can still cause interference between multiple applications, as they share a memory-side router.

Figure 5 shows interconnect latency for latency-critical applications (`LLaMA3` and `GPT-2`) when co-running with best-effort workloads in GPGPU-Sim [7, 10, 88] (detailed in §6). Flit injection latency is the time from memory request arrival at the interconnect to injection into the router through an input port (Figure 4 ❶). Output VC allocation latency is the time from flit arrival at the router to its ejection through the output port (Figure 4 ❷). Both latencies for latency-critical applications increase by 65% and 213% on average due to co-running tenants. This occurs because output ports connected to L2 cache partitions are shared, so `LLaMA3` or `GPT-2` flits stall when output port VCs are already occupied. Although each application has a dedicated input port, VC allocation stalls create backpressure that increases input port buffer occupancy and causes additional stalls. As a result, most applications, except compute-bound ones (e.g., `MM`), interfere with latency-critical applications and degrade their throughput.

3.4 Interference on Memory Subsystems

Queues between the interconnect and both the L2 cache and memory controller (Figure 2) are essential for buffering. When multiple

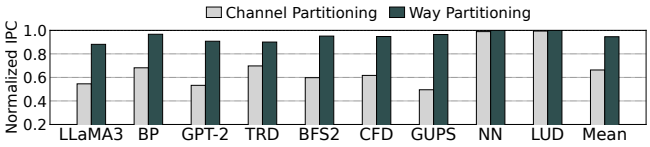


Figure 7: Performance comparison between channel partitioning and way partitioning.

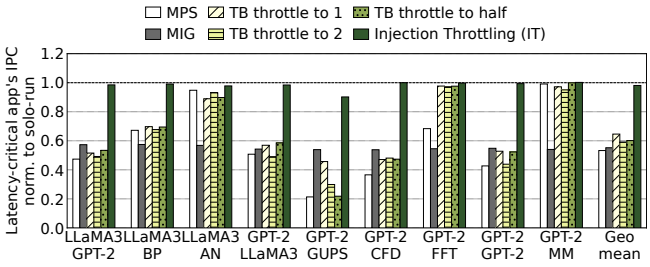


Figure 8: Performance of latency-critical applications (first row) while modulating active threadblocks from the tenant. Although we reduce the number of active threadblocks to 1, tenants still interfere with latency-critical applications' performance.

applications compete for these queues, requests can stall if the queues are full. The left bar in Figure 6 shows the request stall ratio for latency-critical applications (`LLaMA3` and `GPT-2`) accessing memory queues while co-running with others. In most cases, both models show high stall ratios. Memory-intensive workloads, such as `CFD` and `GUPS`, generate a large volume of requests, increasing queuing delay by 46% compared to running alone. This contention harms performance, throughput, and fairness.

We measure simulated L2 cache MPKI (misses per kilo-instruction) and IPC (instructions per cycle) for all benchmark applications (Table 4), both in isolation and in concurrent mixes (Figures 13 and 15). Concurrency causes a wide range of impacts on MPKI and IPC due to contention: IPC drops by up to 66%, and MPKI increases by up to 92%. All mixes suffer performance loss as applications evict each other's data from the shared L2 cache, an effect also observed by others [5, 6, 8, 38, 45, 47, 66, 68, 106].

4 Design Space Analysis

In this section, we analyze the design space for enabling performance isolation on shared resources, such as on-chip interconnects and memory subsystems.

Channel (Vertical) vs. Way (Horizontal) Partitioning. *Vertical partitioning* dedicates slices of the interconnect, cache banks, and memory channels to each application, eliminating contention. This is the approach used by MIG. Because applications rely on bandwidth to increase throughput, most experience degraded IPC when running with a reduced number of channels. MIG also partitions L2 cache banks, which are directly connected to each memory controller, reducing both memory bandwidth and capacity. We found that *horizontal partitioning* via way partitioning [18, 84, 85, 95] controls cache allocation without limiting bandwidth. Multiple applications are assigned to different cache ways in each bank, preserving channel-level parallelism while preventing them from evicting each other's data. Figure 7 compares channel and way partitioning, normalized to a baseline with full access to L2 capacity. Channel partitioning cuts half of the channels, and way partitioning removes half of the ways, yielding *equal capacity* for both.

> ***Insight #1:*** Way partitioning is strictly better for performance, showing 5% performance degradation on average, while channel partitioning shows 33% degradation.

Prioritizing Memory Requests. Hardware channel partitioning allocates both cache capacity and memory bandwidth, while way partitioning mediates only cache capacity, leaving applications vulnerable to unfair access to memory bandwidth. When memory-intensive applications issue many requests, bandwidth-sensitive ones can be slowed. Memory request prioritization is an alternative mechanism for mediating access to the interconnect and memory system. It can increase an application's share of bandwidth by placing its requests into a high-priority queue that is always served first. Prioritizing requests can reduce queuing delay and increase effective memory bandwidth. An efficient design could combine way partitioning with request prioritization to provide fair access to both resources without incurring the performance penalty of channel partitioning.

> ***Insight #2:*** Combining way partitioning with request prioritization mediates contention for both capacity and bandwidth without the risk of under-utilization inherent in vertical partitioning (MIG).

Virtual Channel Partitioning. Memory request prioritization and cache partitioning reduce interference in the memory subsystems, but contention in on-chip interconnect routers persists. As shown in Figure 5, interconnect traversal latency increases significantly due to virtual channel (VC) allocation failures. In such cases, VC partitioning mitigates contention by dedicating specific VC ranges to different applications. For fairness, an ideal policy would partition VCs evenly so that each application receives a fair share. However, when one application generates most of the traffic on a shared router and consumes a disproportionate number of VCs, it becomes necessary to restrict its VC allocation more aggressively. Just as importantly, enabling VC partitioning at the right time is critical, since unnecessary partitioning can lead to VC under-utilization.

> ***Insight #3:*** VC partitioning is beneficial for both asymmetric priority scenarios and symmetric priority scenarios (fairness). However, it is essential to determine the appropriate timing for enabling VCP and how to distribute VCs across applications.

Injection Throttling. The aforementioned techniques mitigate interference, but they may *fail to generate sufficient backpressure* on best-effort applications, allowing them to continue contending with latency-critical workloads. Prior approaches, such as SM throttling [64] or thread-level parallelism modulation [53, 87], reduce memory traffic from best-effort applications, but they still degrade the performance of latency-critical workloads. Figure 8 shows GPT-2's normalized IPC when the number of active threadblocks for the tenant application is reduced. While modulating active threadblocks

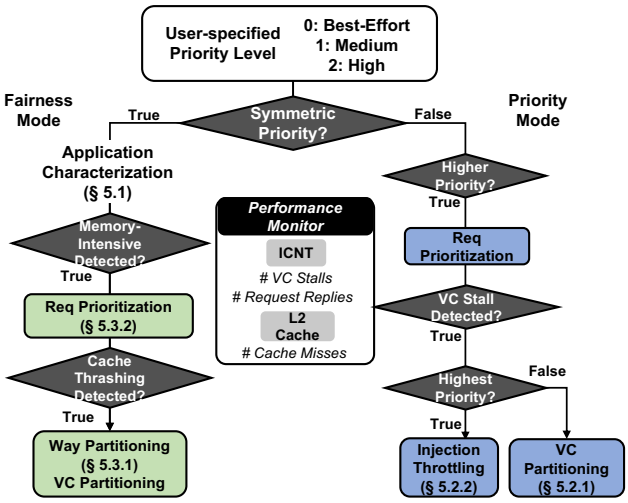


Figure 9: Flow chart with *ESA* isolation components.

lowers memory traffic, GPT-2 performance consistently degrades. Even with only a single threadblock activated (`TB throttling to 1`) for tenant SMs, interference on shared resources persists, impacting GPT-2 performance.

The key limitation is that threadblock throttling controls execution indirectly rather than controlling memory traffic at the point where interference occurs. Even a small number of active threadblocks can continue generating enough requests to occupy shared interconnect buffers and memory-system resources. We instead control memory traffic before requests enter the interconnect by throttling injections at the SM side. We call this injection throttling (*IT*). Because *IT* acts before requests occupy shared buffers, it can directly suppress the source of interconnect contention without reducing thread-level parallelism when the interconnect is uncongested. Figure 8 shows that *IT* preserves prioritized workload performance with only 1.2% loss, while still allowing best-effort applications to make forward progress (Figure 14). However, *IT* must be applied carefully to avoid unnecessary over-throttling (§7.3.2).

> ***Insight #4:*** Injection throttling is an effective mechanism for protecting latency-critical workloads: by controlling contention at the point of injection, it can dedicate bandwidth to prioritized tenants while still allowing best-effort tenants to make progress.

Summary. No single isolation technique addresses contention across the entire memory hierarchy. Application mixes can stress multiple resources, and isolating one can leave residual contention or backpressure elsewhere. Effective GPU sharing therefore requires composing mechanisms rather than applying them independently, activating only the subset needed to isolate contended resources while preserving elasticity when contention is low.

5 *ESA* Design

To provide performance isolation for prioritized applications and ensure fairness across multiple tenants, we present *ESA*, featuring four mediation mechanisms with modest hardware cost. Based on user-defined priorities (Figure 9), *ESA* supports priority-proportional sharing and operates in two modes: Fairness Mode (*FM*) and Priority Mode (*PM*).

The two modes differ in both their objective and the aggressiveness of resource mediation. For fair sharing among applications with equal priority, *FM* does not assume that one application should dominate another. Instead, it uses application characterization from a Performance Monitor (*PerfMon*) to identify *cache-thrashing* or *memory-intensive* applications and selectively activates mechanisms that redistribute shared cache capacity, interconnect access, and memory bandwidth. *FM* activates virtual channel partitioning (§5.2.1), way-based cache partitioning (§5.3.1), and memory request prioritization (§5.3.2) only when such behaviors are detected.

In the case of asymmetric priorities among applications, *ESA*'s *PM* protects higher-priority applications more aggressively. *PM* activates congestion detection on the interconnect router and enables prioritization mechanisms such as virtual channel partitioning and memory request prioritization. For applications assigned the highest priority (value 2), *PM* employs injection throttling (§5.2.2) on the interconnect, suppressing interfering traffic before it occupies shared router resources. Thus, *ESA* progressively applies stronger isolation as the required level of performance protection increases, while preserving elastic sharing when contention is low.

Determining User-specified Priority. Users can specify a priority for each process to maintain fairness or prioritize one application with robust performance isolation. We provide three priority levels (0 to 2), making it simple to differentiate between multiple applications. When more than three applications run concurrently, all except the lowest-priority one are prioritized by the *PM*. While NVIDIA allows kernel-level prioritization via `cudaStream` [72], it applies only within a single application. In contrast, *ESA* enables application-wide prioritization, where priority levels govern not only scheduling but also hardware resource mediation. This allows all kernels of a high-priority application to be favored over others.

5.1 Application Behavior Characterization

Fairness Mode (*FM*) must identify each application's behavior to selectively apply techniques only when necessary, ensuring fairness and performance isolation. *ESA* detects *cache-thrashing* using *PerfMon*, which counts L2 misses and instructions per application while differentiating applications using the address space identifier (ASID) of each request. *PerfMon* flags an application as *cache-thrashing* if its MPKI exceeds a predefined threshold of 10.[2] If the MPKI exceeds the threshold, *PerfMon* records the ASID and sets a bit in its table to classify it as cache-thrashing.

To detect *memory-intensive* behavior, *ESA* observes memory request replies from the L2 cache to the interconnect. *PerfMon* compares each application's bw_ratio ($RequestReplies_i$ / $\sum_{j=1}^{numApps} RequestReplies_j$) to its fair share. If App_i exceeds its fair share, it is classified as memory-intensive. These two signals capture complementary forms of interference: MPKI identifies applications that cause destructive cache contention, while the request-reply ratio identifies applications consuming a disproportionate fraction of shared memory bandwidth.

[2]We examine the distribution of MPKI across all 25 of our workloads and find a clear inflection point near 10.0.

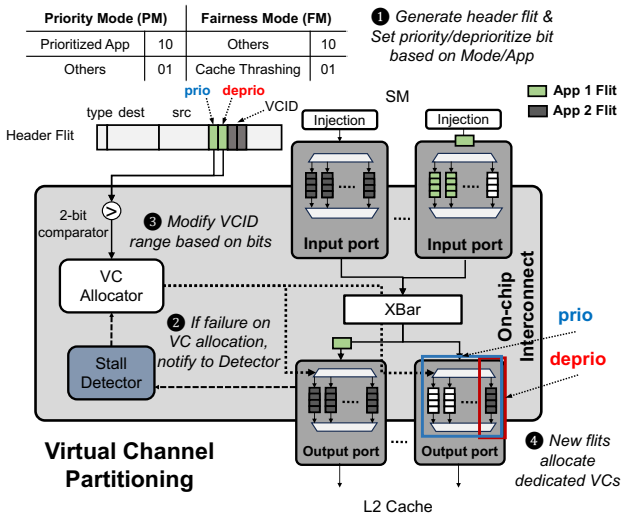


Figure 10: *ESA*'s virtual channel partitioning (Priority Mode and Fairness Mode).

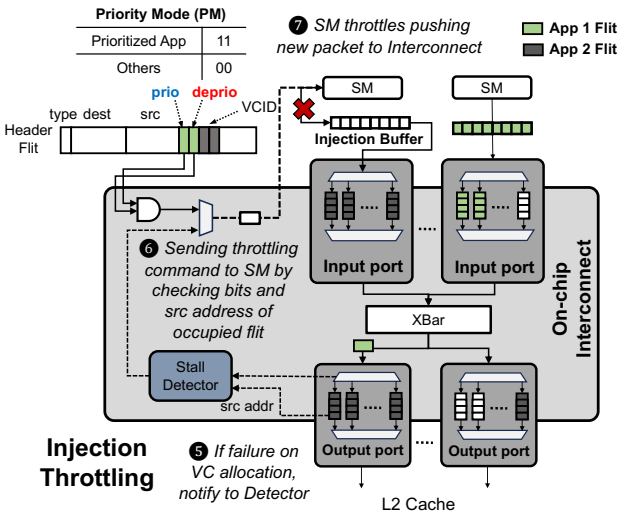


Figure 11: *ESA*'s injection throttling (Priority Mode).

Upon detecting a cache-thrashing or memory-intensive application, *ESA* determines which application should be prioritized for the next epoch (epoch length sensitivity is studied in §7.3.1). As these applications impact other tenants through cache evictions and memory queue stalls (§3.4), it is essential to apply techniques to alleviate the contention they cause.

5.2 On-chip Interconnect Isolation

5.2.1 Virtual Channel Partitioning. As described in §4, virtual channel partitioning (VCP) effectively mitigates slowdowns caused by sharing the on-chip interconnect. Both Fairness Mode (*FM*) and Priority Mode (*PM*) can activate VCP. *ESA* determines which mode to enable based on the user-specified priority value: *FM* is activated for symmetric priorities, while *PM* is used for asymmetric priorities.

In *FM*, VCP is enabled if a cache-thrashing application is detected. In *PM*, VCP is enabled unless a highest-priority application is present, in which case additional isolation techniques are applied (see §5.2.2). Both modes use two bits in the header flit: a prioritizing (`prio`) bit and a de-prioritizing (`deprio`) bit (Figure 10 ❶). In *PM*, prioritized applications set the `prio` bit (10) to access all VCs, while best-effort tenants set the `deprio` bit (01) to use fewer VCs, thereby avoiding interference. We allocate two VCs for best-effort tenants to separately handle request and reply flits. In *FM*, *ESA* sets the `deprio` bit (01) for cache-thrashing applications and the `prio` bit (10) for others, allowing non-thrashing applications to share VCs without interference.

After setting the bits, if a flit cannot allocate an output port's VC due to unavailability, the Stall Detector is notified ❷. It checks the `prio` and `deprio` bits to decide whether to activate VCP. If VCP is required, it signals the VC allocator to adjust available VCs per application based on these bits. If the `deprio` bit is set to 1, the VC allocator restricts available VCs to two by modifying the VCID in the header flit ❸. If the `prio` bit is set to 1, VCs are allocated based on flit type. Ultimately, *ESA* partitions VCs via the VC allocator across all applications, ensuring prioritized access for high-priority tenants ❹.

5.2.2 Injection Throttling. Although VCP provides isolation during VC allocation, latency-critical applications may still suffer performance degradation, potentially impacting tail latency. To eliminate performance interference, *ESA* employs Injection Throttling (*IT*) in Priority Mode (*PM*).

When a flit stalls due to VC allocation failure, a counter tracks the cycles for which the flit stalls attempting to allocate a VC. If it exceeds a predefined threshold (§7.3.2), the Stall Detector is notified (Figure 11 ❺). The *IT* decision module then compares the bits in the flit header to determine if it belongs to the highest-priority application. If the flit requires prioritization, *IT* sends a throttling command to the originating SM by checking the source ID of the flit already occupying the VC ❻. After receiving the command, the SM halts sending memory packets to the interconnect's injection buffer, effectively ceasing flit injection to the router for a predefined duration ❼.

5.3 Cache and Memory Isolation

5.3.1 Way-based Partitioning. When an application's behavior is characterized as *cache-thrashing* by *PerfMon*, *ESA* partitions cache ways to ensure balanced allocation for each application in both Fairness and Priority Modes (Figure 12 ❶). It employs a bitmask for way allocation, specifying the range of allocated ways. It maps the ASID of each memory request to the designated way index range, routing requests accordingly. This approach effectively prevents excessive evictions caused by cache-thrashing applications.

5.3.2 Memory Request Prioritization. To mediate bandwidth (§3.4), *ESA* uses two logically separated memory queues with different priorities, guided by application behavior. *PerfMon* (Figure 12) identifies *cache-thrashing* (high L2 MPKI) and detects *memory-intensive* applications (§5.1) based on bandwidth usage. It refreshes memory request statistics and prioritizes applications in each epoch, whose length is empirically derived (§7.3.1). *PerfMon* sets a *priority tag* on

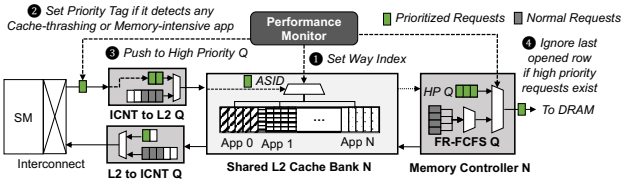


Figure 12: *ESA* performance isolation architecture for the memory subsystem.

memory requests from applications that are neither *cache-thrashing* nor *memory-intensive* ❷. Untagged requests go to the normal queue, while tagged requests are directed to the high-priority queue ❸. Previously queued requests from the same application must be drained before new high-priority requests can be enqueued.

Instead of physically separating memory queues to support high-priority requests, *ESA* uses Dynamically Allocated Multi-Queue (DAMQ), a technique also used in on-chip network design [98]. DAMQ enables physically shared but virtually separated queues by maintaining separate virtual linked lists, each corresponding to a different priority. These linked lists allow high-priority memory requests to be selected within a single queue. This approach effectively partitions the queue without additional hardware overhead or under-utilized queue buffers. However, high-priority requests may stall if all entries are full, particularly between the interconnect and L2 cache (*icnt_L2* in Figure 6). To address this, *ESA* reserves a buffer at *icnt_L2* queues to temporarily hold stalled high-priority requests, thereby mitigating latency overheads.

ESA's FR-FCFS scheduler in the memory controller is also priority-aware. By default, FR-FCFS preferentially dequeues requests that match the currently active row. *ESA* modifies this policy such that priority takes precedence over row address matches ❹. If prioritized requests exist in the queue, the modified scheduler chooses a prioritized request with the same row address if one exists; otherwise, it chooses from the head of the prioritized queue.

6 Methodology

We model the NVIDIA Ampere architecture [22] by modifying Mosaic GPGPU-Sim [7, 10, 88] v3.2.2 to evaluate *ESA*, MIG, and MPS with multi-tenancy support. We use the NVIDIA A100 GPU configuration as a baseline (Table 2) and adopt the cache/MSHR configuration from GPGPU-Sim 4.0 [54]. Due to the limited working set size of the simulated workloads, we evaluate both 10MB and 40MB cache sizes. The 10MB configuration exposes contention patterns across a broader range of workload mixes, while the 40MB configuration matches the capacity of the real A100 hardware and verifies that *ESA*'s qualitative behavior persists under a realistic cache size. For on-chip interconnect isolation (*VCP* and *IT*), we extend BookSim [46].

MIG Configuration. When the A100 GPU operates in MIG mode, SM partitions must follow the configuration in Table 3, i.e., one GPU is divided into up to 7 GPU slices, each with 14 SMs, 10 L2 cache banks, and 10 memory channels [22, 23]. Since MIG partitions are asymmetric, we include a MIG_ideal configuration. *ESA* and MIG_ideal both split resources evenly, while MIG adheres to Table 3 constraints.

GPU Core Configuration	
System Overview	108 cores, 64 execution units per core.
Shader Core	765 MHz, 9-stage pipeline, 64 threads per warp, LRR scheduler.
Cache and TLB Configuration	
Private L1 Cache	128KB, 256-way associative, LRU, L1 misses are coalesced before accessing L2, 1-cycle latency.
Private L1 TLB	128 entries per core, fully associative, LRU, 1-cycle latency.
Shared L2 Cache	10MB/40MB total, 32-way associative, LRU, 80 cache banks, 2 ports per bank, 10-cycle latency, 192 MSHR entries per bank.
Shared L2 TLB	1024 entries total, 16-way associative, LRU, 2 ports, 10-cycle latency, 3 µs TLB shootdown latency.
Interconnect and Memory Configuration	
Interconnect	765 MHz, 4-stage pipeline, Crossbar, Flit size 32, 16 VCs, 16 Buffers per VC Islip VC/Switch allocator
Memory Queue	64 entries for each queue.
DRAM	HBM2 1215 MHz [73, 100], 80 channels, 16 banks, FR-FCFS scheduler [86, 133], burst length 2.
Page Table Walker	Shared page table walker, 4-level page tables.
Page Walk Cache	16-way 8KB, 10-cycle latency.

Table 2: Configuration of the simulated system.

# of app	MIG
2 apps	2 × 3 GPU slices
3 apps	1 × 3 GPU slices, 2 × 2 GPU slices
4 apps	3 × 2 GPU slices, 1 × 1 GPU slice
6 apps	1 × 2 GPU slices, 5 × 1 GPU slice

Table 3: MIG configuration for different GPU instances.

Application Behavior	Application Name
Compute Bound	**AlexNet(AN)**, **ResNet(RN)**, **NN**, **BP**, FWT, LPS, 3DS, RAY, HS, MM, LIB, BLK, LUD, JPEG, FFT
Memory Intensive	**LLaMA-3.1 8B (LLaMA3)**, **GPT-2**, LUH, SCP, ATAX, MVT, CONS, CFD
Cache Thrashing	GUPS, BFS2

Table 4: Workload characterization under contention.

Workloads. We select 25 applications from Tango [49], LLaMA 3.1 [31], and GPT-2 [4, 12] for computer vision and LLM inference, and Polybench [33], CUDA SDK [71], Rodinia [14], Parboil [94], LULESH [50, 51], and SHOC [24] suites for GPGPU applications. Table 4 lists the selected applications and their behavior characterization under contention. To manage long simulation times, we limit each application's execution to the first 1 billion instructions. Applications are restarted as needed to ensure all applications run concurrently until they each reach the 1 billion instruction threshold.

7 Evaluation

7.1 Performance Results

We evaluate the performance, elasticity, and isolation of *ESA* against a number of design variants of *ESA* as well as *MIG*. Our baseline is a model of *MPS* [74], which spatially partitions SMs but allows unrestricted sharing of interconnect, L2 cache, and memory channels. Other baselines include *MIG* and *MIG_ideal*, each of which

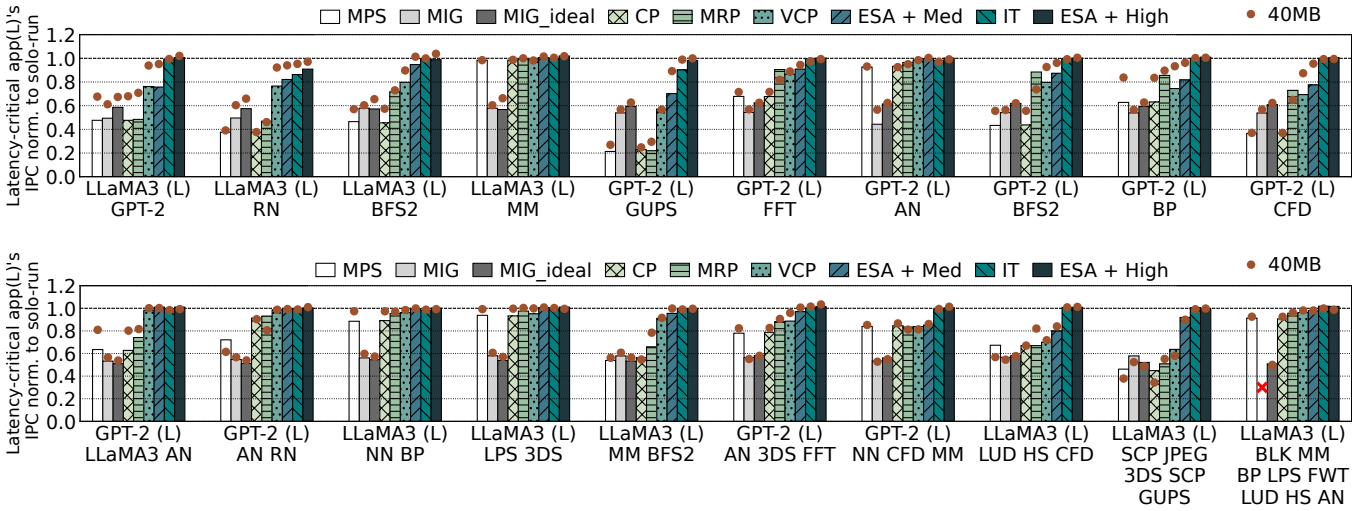


Figure 13: IPC of the latency-critical application (L), normalized to its solo-run IPC, under *Priority Mode* (Higher is better).

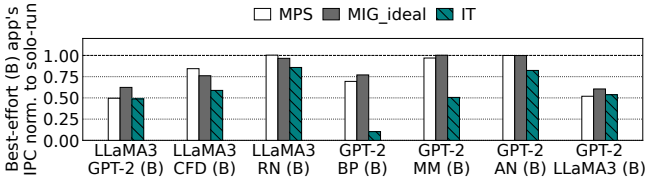


Figure 14: IPC of the best-effort application (B) (second row), normalized to its solo-run IPC (Higher is better).

hard-partitions all the resources, including both SMs and memory subsystems. *MIG* assigns SMs in accordance with the real hardware (Table 3). *MIG_ideal* evenly partitions SMs and memory.

Based on the priority among applications, we activate either Fairness Mode (*FM*) or Priority Mode (*PM*) (§5). For *FM*-enabled scenarios, we evaluate the individual component performance of *ESA*: *Way-based Cache Partitioning* (*CP*), *Memory Request Prioritization* (*MRP*), *Virtual Channel Partitioning* (*VCP*), *CP+MRP* (which includes *CP* and *MRP*), and *ESA-Fair* (including all components). For the *PM*-enabled scenarios, we evaluate the individual component performance of *CP*, *MRP*, *VCP*, and *Injection Throttling* (*IT*). We also evaluate *VCP+CP+MRP* and *IT+CP+MRP* to compare two different interconnect prioritizations with memory isolation techniques.

7.1.1 Priority Mode Results. Figure 13 shows the average normalized IPC of latency-critical applications (`GPT-2(L)` and `LLaMA3(L)`) when co-running with best-effort applications. The results include the same mixes with a 40MB L2 cache, matching real A100 hardware. The larger cache reduces contention, evidenced by a 14% drop in MPKI for `LLaMA3(L)_GPT-2`, which lowers *ESA*'s performance gain by 19%. However, overall trends across all mixes remain similar to those observed with the 10MB cache.

We choose best-effort tenants from ML training and GPGPU applications that exhibit diverse performance characteristics. `LLaMA3(L)` and `GPT-2(L)` experience significant performance drops on *MPS* across all mixes due to contention. *MIG* and *MIG_ideal*, which strictly partition hardware, also degrade performance, as both `LLaMA3(L)` and `GPT-2(L)` rely heavily on memory parallelism, and channel partitioning limits available bandwidth.

Without *ESA*, latency-critical application performance varies with the characteristics of the best-effort tenants. For example, `LLaMA3(L)` performance drops by 52% and 62% when co-running with memory-intensive tenants like `GPT-2` and `BFS2` on *MPS*. In contrast, `GPT-2(L)` sees less than an 8% drop when paired with compute-intensive workloads like `AN`. Regardless of the tenants' performance behavior, *ESA*'s four prioritization techniques effectively mitigate interference, delivering near-native performance for both `LLaMA3(L)` and `GPT-2(L)` under the highest priority setup.

ESA prioritizes latency-critical applications using *IT* while ensuring forward progress for best-effort tenants. Figure 14 shows the IPC of best-effort tenants (B) when *IT* is enabled. *ESA* predominantly activates *IT* for `LLaMA3` or `GPT-2` when it detects stalls in VC allocation on the interconnect. As a result, best-effort tenants experience performance drops ranging from 13% to 89%, but *still make progress* while *IT* dedicates bandwidth to latency-critical applications.

Under medium priority, *ESA* enables *VCP*, *CP*, and *MRP*, delivering 17% higher performance than *MPS*. *VCP* is especially effective when tenants show memory-intensive behavior, generating high request volumes that disrupt VC allocation. For example, in the `LLaMA3(L)_GPT-2` and `GPT-2(L)_GUPS` mixes, *VCP* outperforms other techniques such as *CP* and *MRP*. *MRP* also proves effective in `GPT-2(L)_BP`, where contention occurs in shared memory queues. By combining *VCP*, *CP*, and *MRP*, *ESA* minimizes performance degradation for latency-critical applications, limiting the average performance drop to just 19% compared to native execution.

We evaluate mixes with up to eight concurrent best-effort applications under *Priority Mode*. *ESA* recovers 42% of the performance loss observed under *MPS* and achieves near-native performance. Across all mixes, performance drops by only 13% under medium priority with *VCP+CP+MRP* and by just 0.2% under high priority with

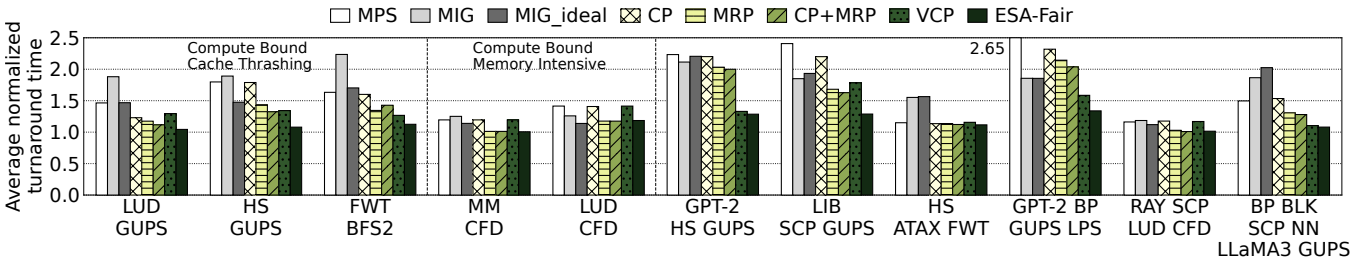


Figure 15: Average normalized turnaround time with *Fairness Mode* (Lower is better).

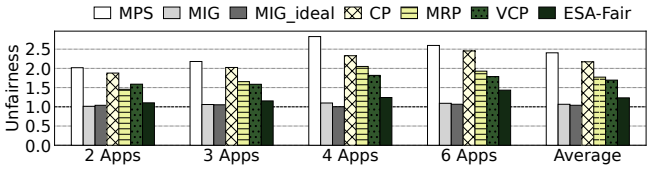


Figure 16: Average unfairness of each application mix under *Fairness Mode*. Results closer to 1 indicate better fairness.

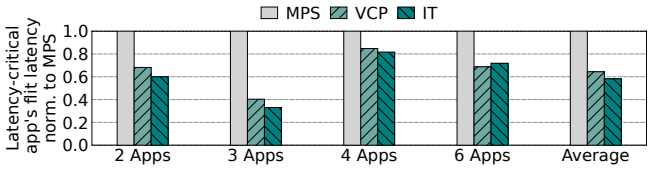


Figure 17: Flit latency with different techniques.

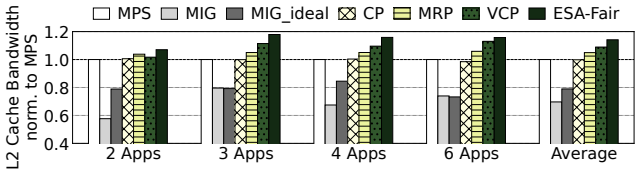


Figure 18: Average L2 cache bandwidth normalized to *MPS*.

IT+CP+MRP. In contrast, *MPS* and *MIG* degrade performance by 31% and 55%, respectively, due to interference, resource fragmentation, and reduced memory parallelism.

7.1.2 Fairness Mode Results. *Fairness Mode* provides fair access to shared resources by minimizing contention and under-utilization. Figure 15 presents the average normalized turnaround time (ANTT), defined as $\frac{1}{n}\sum\frac{Cycles_{Shared}}{Cycles_{Alone}}$, which captures both system performance and fairness. We separate the two-app mixes according to the behavior characterization from §5.1. Mixes involving a cache-thrashing application experience significant performance penalties in *MPS* due to contention. In `LUD_GUPS` and `HS_GUPS` with *MPS*, compute-bound applications suffer performance loss because their datasets are evicted from the cache, and memory bandwidth is consumed by thrashing applications. *ESA* mitigates these issues, reducing execution time.

In the *MIG_ideal* case, each application uses dedicated L2 cache banks and memory partitions, preventing performance degradation from cache-thrashing and memory-intensive applications. However, *MIG* imposes an ANTT penalty for applications sensitive to memory parallelism. For instance, `FWT_BFS2` and `HS_ATAX_FWT` on *MIG* and *MIG_ideal* show worse performance compared to both *ESA* and *MPS*. *MIG*'s strict SM and memory partitioning restricts memory parallelism, degrading performance even when other resources are under-utilized. By contrast, *ESA* does not restrict access to memory partitions and only isolates performance when there is contention.

ESA outperforms *MPS* and *MIG* for all workload mixes. Each technique's effectiveness varies with application characteristics. For instance, in the `HS_GUPS` mix, *VCP* outperforms *MRP* and *CP* because it manages overloaded interconnect traffic from cache-thrashing applications by limiting the number of available VCs. Conversely, *MRP* provides better performance isolation in mixes such as `LUD_CFD`, which have contention on memory queues due to the memory-intensive behavior of `CFD`. *MRP* ensures fair access to memory bandwidth by prioritizing `LUD`'s requests, balancing memory accesses between the applications.

We increase the number of concurrent applications up to six. Even in these six-application mixes, where contention is highest, *ESA* delivers up to 33% higher throughput than *MPS* and 20% higher than *MIG_ideal*; the larger gains reported in §7.1 come from mixes with fewer, more contention-sensitive co-runners.

Figure 16 shows the average unfairness ($Max\frac{IPC_{Alone}}{IPC_{Shared}}$ [5, 26, 32, 56]) of each application mix under *Fairness Mode*. Results closer to 1 indicate better fairness. As expected, *MIG* and *MIG_ideal* provide the best fairness across all mixes by strictly partitioning resources, ensuring performance closely matches standalone execution. *ESA* shows its worst-case unfairness under six concurrent applications due to higher interference. However, *ESA* still reduces unfairness by up to 48% compared to *MPS*, thanks to its integrated techniques, demonstrating *ESA*'s ability to better balance performance, utilization, and isolation than *MIG*.

7.2 Component Analysis

7.2.1 Isolation for On-chip Interconnect. *VCP* and *IT* enable prioritization and manage contention by partitioning the number of available VCs or throttling flit injection on the interconnect. Figure 17 shows the average flit end-to-end latency for latency-critical applications (`LLaMA3 (L)` and `GPT-2 (L)`) when running alongside best-effort applications. We normalize latency results from each technique to the *MPS* baseline. We observe that both *VCP* and *IT* can effectively prioritize latency-critical applications' flits as soon as they detect delays from stalls. Therefore, these techniques reduce the end-to-end flit latency by 34% and 42%, respectively, compared to *MPS*.

7.2.2 Isolation for Memory Subsystem. In Figure 15, mixes including LUD_GUPS and LIB_SCP_GUPS show that using *CP* (way partitioning) improves performance. Compute-bound applications such as LUD from LUD_GUPS see L2 cache MPKI drop by 93% and IPC increase by 32%. Not all applications (e.g., GUPS) benefit from way partitioning: if an application thrashes the L2 cache, way partitioning reserves cache space that will not help its performance.

Figure 15 shows that *MRP* provides better performance in most cases where thrashing applications are running together. Prioritization improves performance isolation where *CP* falls short. For example, in the LUD_CFD mix, LUD's memory requests can be delayed by the many memory requests from CFD. In this case, *MRP* prioritizes LUD's requests after detecting unfairness.

We measure L2 bandwidth and compare it across mixes of concurrent applications in Figure 18. While MIG guarantees private memory channels and queues to each application, overall L2 bandwidth is lower than with *MPS* or *ESA*. Since memory-intensive applications have limited channel access with MIG, their performance is reduced. On the other hand, *ESA* increases L2 bandwidth by up to 51% by enabling applications to utilize the full aggregate L2 bandwidth.

7.3 Sensitivity Analysis

7.3.1 Sensitivity to Epoch Length for PerfMon. ESA refreshes memory request statistics and prioritizes applications in each epoch, whose length is empirically derived. We evaluate epoch lengths from 5K to 1000K cycles and find that performance peaks at 10K. *ESA* uses a length of 10K to balance responsiveness against the overhead of over-modulation observed at 5K.

7.3.2 Sensitivity to IT *Activation Cycles.* For the highest-priority application, *ESA* activates *IT* upon detecting flit stalls on the interconnect, with a short delay to avoid over-throttling best-effort tenants. We evaluate LLaMA3(L) and GPT-2(L) with varying *IT* activation delays (10–1000 cycles) and observe that longer delays degrade prioritized performance (e.g., GPT-2_CFD mix). A 10-cycle delay provides the best performance for the prioritized application.

7.4 Overhead Analysis

7.4.1 Area Overhead. ESA's *PerfMon* uses two 4-byte counters per application to track L2 cache misses and instruction counts, and one additional 4-byte counter for L2-to-interconnect replies. The Stall Detector employs a 4-bit counter per ejection router to count flits from VC allocation stalls. Supporting up to six concurrent applications requires 72 bytes for *PerfMon* and 60 bytes for the Stall Detector across all routers. Behavior characterization adds just 2 bits per application.

Enabling *VCP* and *IT* requires 2 additional bits in the header flit, while memory request prioritization adds a 1-bit tag per request. As described in §5.3.2, a buffer is added between the interconnect and L2 cache partition to hold requests when the queue is full. For high-priority queues, including the FR-FCFS queue in the memory controller, *ESA* leverages existing mechanisms like DAMQ [98], avoiding extra buffer cost. However, *ESA* adds a temporary buffer for stalled high-priority requests in *icnt_L2* queues to reduce latency overhead. We analyze it using the DSENT tool [96] and find that *ESA* incurs a 5.3% area overhead over the baseline design.

7.4.2 Power Analysis. We evaluate *ESA*'s energy and power using GPUWattch [62] on the validated V100 model from Accel-sim [54]. *ESA* reduces overall energy consumption by 5% compared to *MPS*, despite consuming 14% more power due to increased interconnect and memory resource utilization. The energy savings stem from faster workload completion.

8 Related Work

Software-based GPU Sharing Techniques. GPU sharing has been widely studied [7–9, 15, 47, 67, 77, 109, 113, 116, 120]. Recent works [61, 92, 99, 123, 124, 127] offer software-based performance isolation by prioritizing latency-critical jobs, but require kernel rewrites for co-scheduling, rely on profilers to capture kernel characteristics, or suffer from under-utilization due to coarse-grained partitioning. Others [9, 113, 116] support isolation by exploiting ML workload characteristics, adaptively limiting resource allocation, or supporting efficient context switching, but still face under-utilization from temporal sharing.

Hardware-based GPU Sharing Techniques. Hardware-level performance isolation for multi-tenancy has been explored in prior work. Zhang et al. [125] propose isolating GPU memory channels via software-level cache coloring, which complements *ESA*. Other studies target shared page walkers [81], memory queues [69], or memory channels [38], but none address on-chip network interference or provide isolation across the full memory hierarchy. Moreover, some works [38, 69, 119] lack memory protection and only support a limited number of concurrent applications.

GPU Virtual Memory. Virtual address translation based on TLBs is performance-critical in GPUs [105]. Prior work explores TLB design for GPUs [7, 21, 79, 80, 126]. *ESA* leverages these results to enable concurrent applications from different protection domains to share a GPU.

On-chip Interconnect. Virtual channel partitioning (VCP) has been used in NoC designs for heterogeneous CPU-GPU architectures [60, 110], primarily to manage bandwidth between CPU and GPU cores. Prior work applied VCP to GPGPU workloads to improve interconnect utilization by analyzing asymmetric traffic patterns [41], but did not address bandwidth preservation for prioritized or concurrent applications. Other studies propose rank-based prioritization for CPU interconnects [26], modify interconnects to control LLC sharing [128, 129], or mitigate covert-channel attacks [2]. *ESA* enhances interconnects to prioritize applications using injection throttling and VCP, complementing these approaches.

GPU Memory and Thread Schedulers. Prior GPU memory schedulers aim to reduce interference [5, 43, 103, 104] and improve efficiency [13, 48, 122]. Others explore threadblock scheduling to enhance performance and reduce contention [107, 108]. Some approaches mitigate thread contention through concurrency and frequency control [90], or by limiting threadblock count [53]. However, these works do not consider inter-application contention, which is the focus of our work.

Cache Management. Morpheus [25] extends the GPU's last-level cache by enabling idle GPU cores for additional cache storage. TAP [59], a TLP-aware mechanism, modifies utility-based cache

partitioning [84] and insertion policies [40] to lower interference between GPGPU and CPU applications. GPU cache partitioning is often adapted from CPU approaches [18, 84, 85, 95]. Selective cache bypassing [6, 16, 17, 45, 65, 66, 117, 118] reduces contention and improves effective cache capacity. Other studies target interference reduction [114] and cache-thrashing mitigation [39, 40, 83].

9 Conclusion

ESA is a GPU architecture designed for priority-proportional sharing with elastic resource management and performance isolation. Under contention, *ESA* dynamically mediates shared resources by modulating cache way allocation, memory request prioritization, virtual channel partitioning, and injection throttling. These mechanisms prioritize latency-critical ML applications while ensuring fairness across co-running workloads. With its priority-based control, *ESA* preserves near-uncontended performance for latency-critical applications when co-located with best-effort workloads, and improves overall throughput by up to 1.67× in fair-sharing scenarios.

Acknowledgments

We thank our anonymous reviewers for their insightful feedback and comments. This research was supported by the U.S. National Science Foundation (NSF) under grants CNS-2006943 and CNS-2326576.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX symposium on operating systems design and implementation (OSDI 24)*. 117–134.

[2] Jaeguk Ahn, Jiho Kim, Hans Kasan, Leila Delshadtehrani, Wonjun Song, Ajay Joshi, and John Kim. 2021. Network-on-chip microarchitecture-based covert channel in gpus. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 565–577.

[3] AMD. 2015. VDI and Virtualized Workstations. http://www.amd.com/en-us/solutions/professional/virtualization.

[4] Andrej Karpathy. 2024. llm.c. https://github.com/karpathy/llm.c.

[5] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H. Loh, and Onur Mutlu. 2012. Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems. In *ISCA*.

[6] Rachata Ausavarungnirun, Saugata Ghose, Onur Kayıran, Gabriel H. Loh, Chita R. Das, Mahmut T. Kandemir, and Onur Mutlu. 2015. Exploiting Inter-Warp Heterogeneity to Improve GPGPU Performance. In *PACT*.

[7] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *MICRO*.

[8] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher Rossbach, and Onur Mutlu. 2018. Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. In *ASPLOS*.

[9] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. *PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications*. USENIX Association, USA.

[10] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. 2009. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*.

[11] Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954* (2024).

[12] Tom B Brown. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[13] Niladrish Chatterjee, Mike O'Connor, Gabriel H. Loh, Nuwan Jayasena, and Rajeev Balasubramonian. 2014. Managing DRAM Latency Divergence in Irregular GPGPU Applications. In *SC*.

[14] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. 2009. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*.

[15] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. 2017. EffiSha: A Software Framework for Enabling Efficient Preemptive Scheduling of GPU. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 3–16.

[16] Xuhao Chen, Li-Wen Chang, Christopher I. Rodrigues, Jie Lv, Zhiying Wang, and W. W. Hwu. 2014. Adaptive Cache Management for Energy-Efficient GPU Computing. In *MICRO*.

[17] Xuhao Chen, Shengzhao Wu, Li-Wen Chang, Wei-Sheng Huang, Carl Pearson, Zhiying Wang, and W. W. Hwu. 2014. Adaptive Cache Bypass and Insertion for Many-Core Accelerators. In *MES*.

[18] Derek Chiou, Prabhat Jain, Larry Rudolph, and Srinivas Devadas. 2000. Application-specific memory management for embedded systems using software-controlled caches. In *Proceedings of the 37th Annual Design Automation Conference*. 416–419.

[19] Eunyeong Cho, Jehyeon Bang, Ranggi Hwang, and Minsoo Rhu. 2026. PASCAL: A Phase-Aware Scheduling Algorithm for Serving Reasoning-based Large Language Models. In *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1–16.

[20] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 199–216.

[21] Jason Cong, Zhenman Fang, Yuchen Hao, and Glenn Reinmana. 2017. Supporting Address Translation for Accelerator-Centric Architectures. In *HPCA*.

[22] NVIDIA Corp. 2020. NVIDIA A100 Tensor Core GPU Architecture In-Depth. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[23] NVIDIA Corp. 2024. NVIDIA Multi-Instance GPU User Guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/.

[24] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) benchmark suite. In *GPGPU*.

[25] Sina Darabi, Mohammad Sadrosadati, Negar Akbarzadeh, Joël Lindegger, Mohammad Hosseini, Jisung Park, Juan Gómez-Luna, Onur Mutlu, and Hamid Sarbazi-Azad. 2022. Morpheus: Extending the Last Level Cache Capacity in GPU Systems Using Idle GPU Core Resources. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 228–244.

[26] Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R Das. 2009. Application-aware prioritization mechanisms for on-chip networks. In *Proceedings of the 42nd annual IEEE/ACM international symposium on microarchitecture*. 280–291.

[27] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2020. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 492–506.

[28] Aditya Dhakal, Philipp Raith, Logan Ward, Rolando P Hong Enriquez, Gourav Rattihalli, Kyle Chard, Ian Foster, and Dejan Milojicic. 2023. Fine-grained accelerator partitioning for Machine Learning and Scientific Computing in Function as a Service Platform. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 1606–1613.

[29] Micah Dowty and Jeremy Sugerman. 2009. GPU Virtualization on VMware's Hosted I/O Architecture. *SIGOPS Oper. Syst. Rev.* 43, 3 (jul 2009), 73–82. https://doi.org/10.1145/1618525.1618534

[30] J. Duato, AJ. Pena, F. Silla, R. Mayo, and E.S. Quintana-Orti. 2010. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*.

[31] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[32] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. 2009. Coordinated Control of Multiple Prefetchers in Multi-core Systems. In *MICRO*.

[33] 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)*. Ieee, 1–10.

[34] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 443–462.

[35] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 539–558.

[36] Alex Herrera. 2014. NVIDIA GRID: Graphics Accelerated VDI with the Visual Performance of a Workstation. NVIDIA White Paper.

[37] Bodun Hu and Christopher J Rossbach. 2020. Altis: Modernizing gpgpu benchmarks. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 1–11.

[38] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. 2019. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 29–41.

[39] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. 2008. Adaptive Insertion Policies for Managing Shared Caches. In *PACT*.

[40] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High Performance Cache Replacement Using Re-reference Interval Prediction (RRIP). In *ISCA*.

[41] Hyunjun Jang, Jinchun Kim, Paul Gratz, Ki Hwan Yum, and Eun Jung Kim. 2015. Bandwidth-efficient on-chip interconnect designs for GPGPUs. In *Proceedings of the 52nd Annual Design Automation Conference*. 1–6.

[42] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, unjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 947–960.

[43] Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. 2012. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. In *DAC*.

[44] Natalie Enright Jerger, Tushar Krishna, and Li-Shiuan Peh. 2017. *On-chip networks*. Morgan & Claypool Publishers.

[45] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. 2014. MRPB: Memory Request Prioritization for Massively Parallel Processors. In *HPCA*.

[46] Nan Jiang, Daniel U Becker, George Michelogiannakis, James Balfour, Brian Towles, David E Shaw, John Kim, and William J Dally. 2013. A detailed and flexible cycle-accurate network-on-chip simulator. In *2013 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 86–96.

[47] Adwait Jog, Onur Kayiran, Tuba Kesten, Ashutosh Pattnaik, Evgeny Bolotin, Niladrish Chatterjee, Stephen W. Keckler, Mahmut T. Kandemir, and Chita R. Das. 2015. Anatomy of GPU Memory System for Multi-Application Execution. In *MEMSYS*.

[48] Adwait Jog, Onur Kayiran, Ashutosh Pattnaik, Mahmut T Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R Das. 2016. Exploiting Core Criticality for Enhanced GPU Performance. In *SIGMETRICS*.

[49] Aajna Karki, Chethan Palangotu Keshava, Spoorthi Mysore Shivakumar, Joshua Skow, Goutam Madhukeshwar Hegde, and Hyeran Jeon. 2019. Tango: A deep neural network benchmark suite for various accelerators. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 137–138.

[50] I. Karlin, A. Bhatele, J. Keasler, B.L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C.H. Still. 2013. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In *IPDPS*.

[51] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. Lulesh 2.0 updates and changes. (2013).

[52] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. 2012. Gdev: First-Class GPU Resource Management in the Operating System. In *USENIX ATC*.

[53] Onur Kayıran, Adwait Jog, Mahmut T Kandemir, and Chita R Das. 2013. Neither more nor less: Optimizing thread-level parallelism for GPGPUs. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE, 157–166.

[54] 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)*. IEEE, 473–486.

[55] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. 2012. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing* (San Servolo Island, Venice, Italy) *(ICS '12)*. Association for Computing Machinery, New York, NY, USA, 341–352. https://doi.org/10.1145/2304576.2304623

[56] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. 2010. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *HPCA*.

[57] Prabhat Kumar, Yan Pan, John Kim, Gokhan Memik, and Alok Choudhary. 2009. Exploring concentration and channel slicing in on-chip network router. In *2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*. IEEE, 276–285.

[58] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[59] Jaekyu Lee and Hyesoon Kim. 2012. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. IEEE, 1–12.

[60] Jaekyu Lee, Si Li, Hyesoon Kim, and Sudhakar Yalamanchili. 2013. Adaptive virtual channel partitioning for network-on-chip in heterogeneous architectures. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 18, 4 (2013), 1–28.

[61] Munkyu Lee, Sihoon Seong, Minki Kang, Jihyuk Lee, Gap-Joo Na, In-Geol Chun, Dimitrios Nikolopoulos, and Cheol-Ho Hong. 2024. ParvaGPU: Efficient spatial GPU sharing for large-scale DNN inference in cloud environments. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[62] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling energy optimizations in GPGPUs. *ACM SIGARCH Computer Architecture News* 41, 3 (2013), 487–498.

[63] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*. PMLR, 19274–19286.

[64] 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*. 49–63.

[65] Chao Li, Shuaiwen Leon Song, Hongwen Dai, Albert Sidelnik, Siva Kumar Sastry Hari, and Huiyang Zhou. 2015. Locality-Driven Dynamic GPU Cache Bypassing. In *ICS*.

[66] Dong Li, Minsoo Rhu, Daniel R. Johnson, Mike O'Connor, Mattan Erez, Doug Burger, Donald S. Fussell, and Stephen W. Redder. 2015. Priority-Based Cache Allocation in Throughput Processors. In *HPCA*.

[67] Teng Li, Vikram K. Narayana, and Tarek El-Ghazawi. 2014. Symbiotic Scheduling of Concurrent GPU Kernels for Performance and Energy Optimizations. In *CF*.

[68] Yun Liang, Xiuhong Li, and Xiaolong Xie. 2017. Exploring cache bypassing and partitioning for multi-tasking on GPUs. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 9–16.

[69] Emir C Marangoz, Kyoung-Don Kang, and Seunghee Shin. 2021. Designing GPU architecture for memory bandwidth reservation. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 87–89.

[70] Kelvin KW Ng, Henri Maxime Demoulin, and Vincent Liu. 2023. Paella: Low-latency model serving with software-defined gpu scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 595–610.

[71] NVIDIA Corp. 2011. CUDA C/C++ SDK Code Samples. http://developer.nvidia.com/cuda-cc-sdk-code-samples.

[72] NVIDIA Corp. 2017. CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/cuda-runtime-api/stream-sync-behavior.html.

[73] NVIDIA Corp. 2021. NVIDIA A100 Tensor Core GPU Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf.

[74] NVIDIA Corp. 2024. Multi-Process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[75] NVIDIA Corporation. 2026. NVML API Reference Guide. https://docs.nvidia.com/deploy/nvml-api/index.html. vR595, last updated April 16, 2026. Accessed: 2026-04-28.

[76] OpenAI. 2021. CLIP ViT-L/14: openai/clip-vit-large-patch14. https://huggingface.co/openai/clip-vit-large-patch14. Hugging Face model card. Accessed: 2026-05-01.

[77] Sreepathi Pai, Matthew J Thazhuthaveetil, and Ramaswamy Govindarajan. 2013. Improving GPGPU concurrency with elastic kernels. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 407–418.

[78] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 118–132.

[79] 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. In *ASPLOS*.

[80] Jonathan Power, Mark D Hill, and David A Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *HPCA*.

[81] B Pratheek, Neha Jawalkar, and Arkaprava Basu. 2021. Improving GPU Multi-tenancy with Page Walk Stealing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 626–639.

[82] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. 2025. Mooncake: Trading More Storage for Less Computation — A KVCache-centric Architecture for Serving LLM Chatbot. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, Santa Clara, CA, 155–170. https://www.usenix.org/

conference/fast25/presentation/qin
[83] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive Insertion Policies for High Performance Caching. In *ISCA*.
[84] Moinuddin K Qureshi and Yale N Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 423–432.
[85] Parthasarathy Ranganathan, Sarita Adve, and Norman P Jouppi. 2000. Reconfigurable caches and their application to media processing. *ACM SIGARCH Computer Architecture News* 28, 2 (2000), 214–224.
[86] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. 2000. Memory Access Scheduling. In *ISCA*.
[87] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. 2012. Cache-Conscious Wavefront Scheduling. In *MICRO*.
[88] SAFARI Research Group. [n. d.]. Mosaic – GitHub Repository. https://github.com/CMU-SAFARI/Mosaic/.
[89] Evan E Schneider and Brant E Robertson. 2015. CHOLLA: a new massively parallel hydrodynamics code for astrophysical simulation. *The Astrophysical Journal Supplement Series* 217, 2 (2015), 24.
[90] Ankit Sethia and Scott Mahlke. 2014. Equalizer: Dynamic tuning of gpu resources for efficient execution. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 647–658.
[91] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. 2025. Dynamollm: Designing llm inference clusters for performance and energy efficiency. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1348–1362.
[92] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 1075–1092.
[93] Foteini Strati, Sara Mcallister, Amar Phanishayee, Jakub Tarnawski, and Ana Klimovic. 2024. D\'ej\aVu: KV-cache Streaming for Fast, Fault-tolerant Generative LLM Serving. *arXiv preprint arXiv:2403.01876* (2024).
[94] J. A. Stratton, C. Rodrigues, I. J. Sung, N. Obeid, L. W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu. 2012. *Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing*. Technical Report IMPACT-12-01. Univ. of Illinois at Urbana-Champaign.
[95] G Edward Suh, Larry Rudolph, and Srinivas Devadas. 2004. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing* 28, 1 (2004), 7–26.
[96] Chen Sun, Chia-Hsin Owen Chen, George Kurian, Lan Wei, Jason Miller, Anant Agarwal, Li-Shiuan Peh, and Vladimir Stojanovic. 2012. DSENT-a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*. IEEE, 201–210.
[97] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. 2014. GPUvm: Why Not Virtualizing GPUs at the Hypervisor?. In *USENIX ATC*.
[98] Yuval Tamir and Gregory L Frazier. 1992. Dynamically-allocated multi-queue buffers for VLSI communication switches. *IEEE Trans. Comput.* 41, 06 (1992), 725–737.
[99] Xiaodan Serina Tan, Pavel Golikov, Nandita Vijaykumar, and Gennady Pekhimenko. 2022. GPUPool: A Holistic Approach to Fine-Grained GPU Sharing in the Cloud. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 317–332.
[100] TechPowerUp. 2020. NVIDIA A100 PCIe. https://www.techpowerup.com/gpu-specs/a100-pcie.c3623.
[101] Aidan P Thompson, H Metin Aktulga, Richard Berger, Dan S Bolintineanu, W Michael Brown, Paul S Crozier, Pieter J In't Veld, Axel Kohlmeyer, Stan G Moore, Trung Dac Nguyen, et al. 2022. LAMMPS-a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Computer Physics Communications* 271 (2022), 108171.
[102] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).
[103] Hiroyuki Usui, Lavanya Subramanian, Kevin Chang, and Onur Mutlu. 2015. SQUASH: Simple QoS-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators. arXiv:1505.07502 [cs.AR].
[104] Hiroyuki Usui, Lavanya Subramanian, Kevin Chang, and Onur Mutlu. 2016. DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators. In *TACO*.
[105] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. 2016. Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems. In *ISPASS*.
[106] Haonan Wang, Fan Luo, Mohamed Ibrahim, Onur Kayiran, and Adwait Jog. 2018. Efficient and fair multi-programming in GPUs via effective bandwidth management. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 247–258.
[107] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. 2015. Dynamic Thread Block Launch: A lightweight execution mechanism to support irregular applications on GPUs. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 528–540. https://doi.org/10.1145/2749469.2750393
[108] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. 2016. Laperm: Locality aware scheduler for dynamic parallelism on gpus. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 583–595.
[109] Zhenning Wang, Jun Yang, Rami Melhem, Bruce R. Childers, Youtao Zhang, and Minyi Guo. 2016. Simultaneous Multikernel GPU: Multi-Tasking Throughput Processors via Fine-Grained Sharing. In *HPCA*.
[110] Hao Wen and Wei Zhang. 2018. Reducing inter-application interferences in integrated cpu-gpu heterogeneous architecture. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 278–281.
[111] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. {MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 945–960.
[112] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. 2023. Beware of fragmentation: Scheduling {GPU-Sharing} workloads with fragmentation gradient descent. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 995–1008.
[113] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent {GPU} sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 69–85.
[114] Carole-Jean Wu and Margaret Martonosi. 2011. Characterization and Dynamic Mitigation of Intra-application Cache Interference. In *ISPASS*.
[115] Yuxing Xiang, Xue Li, Kun Qian, Yufan Yang, Diwen Zhu, Wenyuan Yu, Ennan Zhai, Xuanzhe Liu, Xin Jin, and Jingren Zhou. 2025. Aegaeon: Effective GPU pooling for concurrent LLM serving on the market. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. 1030–1045.
[116] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. *AntMan: Dynamic Scaling on GPU Clusters for Deep Learning*. USENIX Association, USA.
[117] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. 2013. An Efficient Compiler Framework for Cache Bypassing on GPUs. In *ICCAD*.
[118] Xiaolong Xie, Yun Liang, Yu Wang, Guangyu Sun, and Tao Wang. 2015. Coordinated Static and Dynamic Cache Bypassing for GPUs. In *HPCA*.
[119] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. 2016. Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming. In *ISCA*.
[120] Tsung Tai Yeh, Amit Sabne, Putt Sakdhnagool, Rudolf Eigenmann, and Timothy G. Rogers. 2017. Pagoda: Fine-Grained GPU Resource Virtualization for Narrow Tasks. In *PPoPP*.
[121] Hangchen Yu, Arthur M. Peters, Amogh Akshintala, and Christopher J. Rossbach. 2020. AvA: Accelerated Virtualization of Accelerators. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 807–825.
[122] G.L. Yuan, A. Bakhoda, and T.M. Aamodt. 2009. Complexity Effective Memory Access Scheduling for Many-Core Accelerator Architectures. In *MICRO*.
[123] Bowen Zhang, Shuxin Li, and Zhuozhao Li. 2024. MIGER: Integrating Multi-Instance GPU and Multi-Process Service for Deep Learning Clusters. In *Proceedings of the 53rd International Conference on Parallel Processing*. 504–513.
[124] Shulai Zhang, Quan Chen, Weihao Cui, Han Zhao, Chunyu Xue, Zhen Zheng, Wei Lin, and Minyi Guo. 2025. Improving GPU Sharing Performance through Adaptive Bubbleless Spatial-Temporal Sharing. In *Proceedings of the Twentieth European Conference on Computer Systems*. 573–588.
[125] Yongkang Zhang, Haoxuan Yu, Chenxia Han, Cheng Wang, Baotong Lu, Yunzhe Li, Zhifeng Jiang, Yang Li, Xiaowen Chu, and Huaicheng Li. 2025. SGDRC: Software-Defined Dynamic Resource Control for Concurrent DNN Inference on NVIDIA GPUs. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 267–281.
[126] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. 2023. TunneLs for Bootlegging: Fully Reverse-Engineering GPU TLBs for Challenging Isolation Guarantees of NVIDIA MIG. (2023).
[127] Wei Zhao, Anand Jayarajan, and Gennady Pekhimenko. 2024. Tally: Non-Intrusive Performance Isolation for Concurrent Deep Learning Workloads. *arXiv preprint arXiv:2410.07381* (2024).
[128] Xia Zhao, Almutaz Adileh, Zhibin Yu, Zhiying Wang, Aamer Jaleel, and Lieven Eeckhout. 2019. Adaptive memory-side last-level GPU caching. In *Proceedings of the 46th international symposium on computer architecture*. 411–423.
[129] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. 2020. Selective replication in memory-side GPU caches. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 967–980.
[130] Xia Zhao, Sheng Ma, Zhiying Wang, Natalie Enright Jerger, and Lieven Eeckhout. 2019. Cd-xbar: A converge-diverge crossbar network for high-performance gpus. *IEEE Trans. Comput.* 68, 9 (2019), 1283–1296.

[131] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670* (2024).

[132] Kan Zhu, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Yufei Gao, Qinyu Xu, Tian Tang, Zihao Ye, et al. 2024. Nanoflow: Towards optimal large language model serving throughput. *arXiv preprint arXiv:2408.12757* (2024).

[133] William K. Zuravleff and Timothy Robinson. 1997. Controller for a Synchronous DRAM That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order. U.S. Patent Number 5,630,096.