FLYT: Transparent and Elastic GPU Provisioning for Multi-Tenant Cloud Services

Santhosh M. Kumar
Sameer Ahmad
Armaan Chowfin
Purushottam Kulkarni
Indian Institute of Technology Bombay

Anand Eswaran
Praveen Jayachandran
IBM Research
Bangalore, India

Abstract

Modern cloud services such as AI inference, video analytics, and scientific computing exhibit highly variable and bursty GPU demand patterns that static provisioning and coarse-grained sharing mechanism struggle to accommodate efficiently. Existing GPU multiplexing approaches, including NVIDIA MPS and MIG, provide limited flexibility in multi-tenant environments, often leading to resource fragmentation, under-utilization, or unpredictable latency. We present Flyt, a transparent, latency-aware GPU orchestration framework for virtualized cloud services. Flyt enables fine-grain runtime scaling of Streaming Multiprocessors (SMs) and breaks the traditional VM–GPUs binding by allowing applications inside a VM to execute on different GPUs over time. This design supports elastic scaling and live inter–node GPU migration without application or guest OS modifications, by virtualizing GPU memory through address translation and enforcing elastic SM execution caps.

An evaluation on heterogeneous GPUs using TorchServe and Rodinia benchmark applications demonstrates that Flyt maintains predictable bounded latency under dynamic workloads while significantly improving GPU utilization compared to static provisioning. In co-located VM–GPU deployments using shared-memory transport, Flyt achieves performance within 12–15% of native execution for most workloads while providing latency isolation and elasticity under contention, demonstrating that elastic SM allocation can maintain latency targets under bursty load without hardware partitioning.

CCS Concepts

• **Computer systems organization → Cloud computing**.

Keywords

GPU virtualization; Elastic GPU provisioning; Multi-tenant cloud systems; Latency-aware resource management; Cloud GPU sharing

ACM Reference Format:
Santhosh M. Kumar, Sameer Ahmad, Armaan Chowfin, Purushottam Kulkarni, Anand Eswaran, and Praveen Jayachandran . 2026. FLYT: Transparent and Elastic GPU Provisioning for Multi-Tenant Cloud Services. In *Proceedings of the 17th ACM/SPEC International Conference on Performance Engineering (ICPE '26), May 04–08, 2026, Florence, Italy*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3777884.3797818

ICPE '26, Florence, Italy

ACM ISBN 979-8-4007-2325-4/2026/05
https://doi.org/10.1145/3777884.3797818

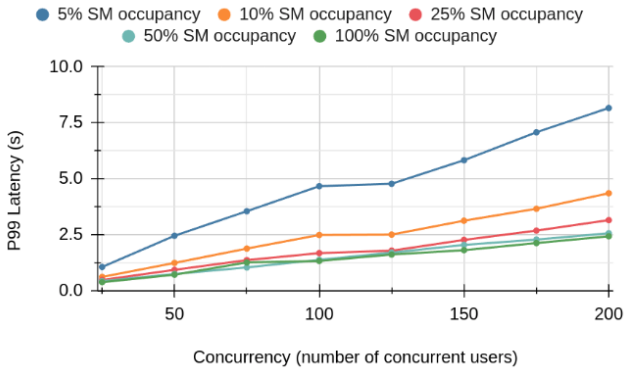


Figure 1: P99 inference latency with increasing workload and static GPU compute allocations.

1 Introduction

The increasing reliance on GPUs has fundamentally reshaped the performance landscape of modern cloud applications, including machine learning (ML) inference, video analytics, and scientific simulations. These workloads often exhibit highly variable and bursty compute demands, making static GPU provisioning both inefficient and costly. Fixed allocation schemes frequently result in significant GPU under-utilization, degrading cost-efficiency and energy proportionality in large-scale deployments [10, 17].

Figure 1 illustrates the limitations of static GPU compute allocation for a representative ML inference workload. For example, with a low provisioning level of 5% of SMs provisioned, doubling concurrency from 50 to 100 users nearly doubles the P99 latency; maintaining comparable latency requires increasing the allocation to approximately 10%. At 200 users, more than 50% of the GPU computation is needed to preserve comparable tail-latency targets. These limitations are particularly pronounced in virtualized cloud environments. GPU-accelerated services are typically deployed inside virtual machines (VMs), where GPU resources are assigned at coarse granularity. In traditional VM-based GPU virtualization, a VM is statically bound to a fixed set of GPU devices, and applications running inside the VM can only execute on those GPUs. As a result, adapting to transient load spikes often requires resizing or migrating the entire VM, even when only a single application needs additional GPU capacity.

Existing GPU sharing mechanisms, such as NVIDIA Multi-Process Service (MPS) [14] and Multi-Instance GPU (MIG) [15] provide partial support for GPU sharing, but remain fundamentally static. MPS enables concurrent kernel execution but does not allow runtime reconfiguration of SM shares, while MIG partitions GPUs into fixed instances that cannot be resized dynamically. Consequently, these approaches lack support for fine-grained, per-application elasticity, and transparent migration in virtualized environments.

To relax the tight coupling between applications, VMs, and physical GPUs, several API-level GPU virtualization frameworks have been proposed. Systems such as vCUDA [18], rCUDA [4], qCUDA [11], and Cricket [5] decouple applications from host-attached GPUs through API remoting, enabling flexible placement and remote execution. However, these systems primarily focus on transparent device access and do not support runtime resizing of GPUs compute or memory, nor the ability for applications inside a VM to scale or migrate transparently across GPUs.

The key challenge in elastic GPU provisioning is that both GPU compute and device memory are traditionally bound to a physical GPU and fixed at context creation. Flyt addresses this challenge by decoupling GPU memory from execution using address translation and dynamically reconfiguring execution contexts with MPS-enforced SM caps. Together, these mechanisms break the VM–GPUs binding at runtime, enabling elastic compute and memory provisioning across GPUs.

To address these limitations, we present Flyt , a transparent GPU virtualization and provisioning framework for elastic resource management in multi-tenant cloud environments. Flyt builds on the Cricket [5] substrate and extends it into a fully elastic and migratable virtual GPU layer. By decoupling application execution from physical GPUs allocation, Flyt enables fine-grained runtime control over GPU compute and memory, supports per-application GPU endpoints, and allows applications inside a VM to scale across multiple GPUs as workload demands change.

Flyt is guided by the following design principles: (i) Transparency; (ii) Elasticity; (iii) SLA Awareness; (iv) Breaking the VM-GPU binding; and (v) Live Migration. The main contributions of this work are:

- Design and implementation of Flyt, an API-level GPU virtualization framework that enables runtime resizing of GPU compute and GPU-agnostic memory virtualization for migration.
- A per-application GPU endpoint abstraction that decouples applications from physical GPUs, enabling transparent live migration and multi-GPU provisioning without VM restart.
- A latency-aware orchestration mechanism that dynamically adjusts GPU allocations based on application-level tail latency.
- An experimental evaluation of Flyt on a multi-GPU cluster using Rodinia benchmarks and production ML inference workloads, demonstrating improved latency adherence and resource efficiency under dynamic load.

By treating SMs as a dynamically allocatable resource decoupled from VM boundaries, Flyt enables latency-aware adaptation under bursty demand without requiring hardware partitioning or application changes. The remainder of this paper is organized as follows: Section 2 reviews the background and related work, Section 3 presents the design and implementation details, Section 4 evaluates the system, and Section 5 concludes.

2 Related work

This section reviews existing techniques for the efficient sharing and utilization of GPU, highlights their limitations, and motivates the need for Flyt.

2.1 Process and API-Level Virtualization

Each GPU application executes within a private CUDA context that encapsulates its execution state, while bindings to compute and memory resources remain effectively static after context creation [13]. As a result, concurrent applications primarily rely on GPU time-slicing, which often leads to reduced utilization and unpredictable latency under contention.

To improve spatial sharing within a single GPU, NVIDIA's MPS enables concurrent kernel execution across processes via static, upfront resource configuration [14]. Several software systems further refine intra-GPU control. Paella applies compile-time and runtime kernel transformations through a specialized runtime to increase utilization [12], while Prism employs a two-level scheduler that dynamically balances space-sharing and time-sharing to meet strict Service Level Objectives (SLOs) [25]. More recently, libsmctrl exposes fine-grained SM control through user-space APIs, enabling explicit SM commitments and placement control [2]. However, these approaches focus on single-GPU, node-local resource management and do not address distributed or multi-GPU settings.

To decouple applications from host-attached GPUs, API-level virtualization frameworks such as rCUDA [4], qCUDA [11], and Cricket [5] intercept CUDA API calls and execute them on remote GPU servers. These systems enable transparent GPU remoting across GPU nodes, but bind applications to fixed remote GPU allocations. Consequently, they lack support for runtime elasticity, per-application scaling, and coordinated GPU resource orchestration across dynamic workloads.

2.2 Hardware-assisted GPU Virtualization

The PCIe pass-through assigns a physical GPU directly to a VM, providing near–bare-metal performance but no sharing capabilities [3]. NVIDIA vGPU extends this model through profile-based partitioning, exposing multiple virtual GPU (vGPU) device endpoints backed by isolated memory regions and time-multiplexed compute execution. This enables predictable multi-VM sharing on a single GPU; however, vGPU profiles are statically defined at initialization, and resizing requires a disruptive GPU reset [16].

The Multi-Instance GPU (MIG) further strengthens isolation by partitioning a GPU into hardware-enforced slices with dedicated compute and memory resources [15]. Although MIG provides strong spatial isolation and performance predictability, partitions are fixed at configuration time and cannot be resized dynamically. Consequently, hardware-level mechanisms trade elasticity for isolation, limiting flexibility for latency-sensitive and dynamically varying cloud workloads.

Table 1: Comparison with GPU provisioning solutions.

System	Capabilities & Characteristics
MPS [14]	Static SM configuration with no runtime SM resizing or migration support.
MIG [15]	Hardware-enforced static partitioning with strong isolation; No runtime resizing support.
rCUDA [4]	Transparent CUDA remoting; no dynamic resource control.
vCUDA [18]	Allows multiple VMs to access host GPUs; No dynamic resource control.
qCUDA [11]	CUDA API virtualization for VMs with GPU attached to host; No dynamic resource control.
Cricket [5]	CUDA API offloading; basic checkpoint/restore; no elastic orchestration.
Paella [12]	Kernel-level scheduling within a single GPU. Does not provide committed GPU resource guarantees.
Orion [20]	Kernel-level scheduling for predictable latency on single GPU. No committed resource guarantees.
AntMan [24]	Cluster-level scheduling for dynamic GPU allocation across jobs; no per-application elasticity or live migration.
Flyt	**SM-level elastic allocation**, **GPU-agnostic memory**, **live migration**, **SLA-aware control**.

2.3 Cluster-Level Orchestration

Cluster-level orchestrators extend GPU management across nodes and applications, coordinating placement and resource allocation at datacenter scale. Systems such as Gandiva [23], AntMan [24], and GParS [22] integrate GPU scheduling into cluster orchestrators to improve utilization and fairness under dynamic workloads. However, these frameworks typically operate above the GPU runtime, relying on static device assignments, fixed profiles, or coarse-grained time slicing. Consequently, they reallocate GPU at the device or partition level, rather than dynamically adjusting per GPU compute or memory for running applications. This separation between cluster-level control and fine-grained GPU execution limits the scope and responsiveness of resource provisioning for latency-sensitive and bursty workloads.

2.4 Limitations of Prior Work

Table 1 summarizes representative GPU virtualization and scheduling systems and highlights the gaps that limit their applicability in elastic cloud environments. Previous approaches fall short in four key dimensions, as discussed in the following.

- **Static provisioning and coarse allocation.** As shown in Table 1, most API-level virtualization systems (rCUDA, vCUDA, qCUDA, and Cricket) bind applications to a fixed GPU endpoint with statically provisioned compute and memory resources. Similarly, hardware-based partitioning mechanisms and kernel-level schedulers (e.g., Paella and Orion) rely on scheduler decisions that cannot be adjusted at runtime. This lack of elasticity leads to under-utilization under bursty workloads and over-provisioning under peak demand.
- **Rigid isolation and limited granularity.** Existing systems primarily operate at coarse granularity—either at the whole-GPU level (rCUDA, vCUDA, qCUDA, Cricket) or at kernel boundaries (Paella, Orion). As indicated in Table 1, none of these approaches supports fine-grained runtime adjustment of GPU compute resources for a running application. As a result, cluster schedulers inherit these rigid allocation units and are unable to reclaim, re-balance, or proportionally reassign GPU compute capacity across active jobs, limiting their ability to respond to dynamic job workload variation.
- **Lack of SLA-aware adaptation.** Most prior systems focus on throughput optimization or fairness at the scheduler level, rather than application-visible performance metrics. None of the compared systems (Table 1) integrate latency-sensitive or SLA-driven control loops, making them ill-suited for latency-critical inference workloads, common in cloud services.

In contrast to these approaches, Flyt shows that combining API-level GPU virtualization with runtime SM resizing and migration (transparent to the application) enables elasticity at a finer granularity than device- or partition-level mechanisms.

3 Flyt Design and Implementation

Flyt enables *runtime* SM resizing, transparent inter-node migration, and feedback-driven latency enforcement, while remaining transparent to applications and preserving the standard CUDA programming model. At the core of Flyt are two mechanisms that decouple GPU resources from device ownership: (i) a GPU-agnostic device-memory abstraction that preserves application-visible pointers across reconfiguration and migration, and (ii) elastic SM execution caps realized via per-context partitioning and context replacement. All higher-level behavior—vertical scaling, live migration, and orchestration—builds on these two mechanisms.

System entities and roles. Flyt comprises (i) *application VMs* running unmodified CUDA programs and hosting the virtualization library and a lightweight VM agent; (ii) *GPU virtualization servers* on GPU nodes that execute CUDA calls on behalf of applications and implement Flyt's core mechanisms; and (iii) a centralized *orchestrator* that places applications, adjusts SM allocations, and triggers migration based on application latency and device telemetry. Unless stated otherwise, both the GPU-agnostic memory abstraction and the elastic SM execution caps are implemented inside the GPU virtualization server.

3.1 Configuration Specification

Flyt exposes a user-facing execution model comprising three logical entities: (i) virtual machines (VMs) that host GPU applications, (ii) GPU servers that execute CUDA kernels, and (iii) a centralized orchestrator that maps applications to GPU servers and dynamically manages resources. Users control the behavior of Flyt through a set of configuration parameters, summarized in Table 2, which are interpreted by the virtualization library, the GPU virtualization server, and the orchestrator. These parameters specify resource limits, latency targets, and placement preferences without exposing the internal mechanisms of Flyt. Configuration is provided at multiple levels: orchestrator, per-VM, per-GPU node (via configuration files) and per-application (via environment variables).

Table 2: Flyt runtime configuration parameters.

Parameter	Description
APP_SM_CORES	Initial SM allocation
APP_GPU_MEMORY	Maximum GPU memory quota (MB/GB)
SCALE_UP_FACTOR	SMs added per scale-up action
SCALE_DOWN_FACTOR	SMs removed per scale-down action
APP_LATENCY_UPPER	Upper latency threshold (ms)
APP_LATENCY_LOWER	Lower latency threshold (ms)
METRIC_SAMPLE_COUNT	Sliding window size for decisions
APP_MODE	VM-dedicated or application-scoped GPU
GPU_AFFINITY_NODES	Preferred GPU nodes and transport mode
GPU_METRIC_INTERVAL	GPU metric sampling interval (s)

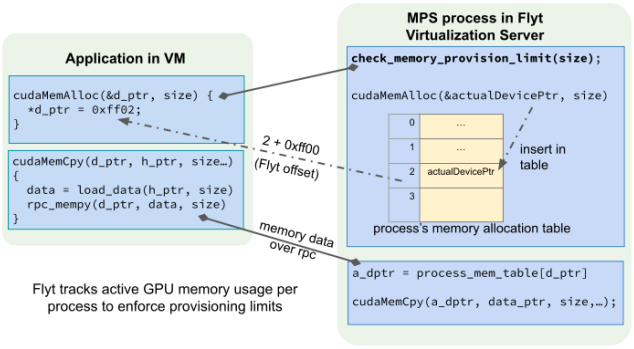


Figure 2: Application-visible device addresses are translated by the GPU virtualization server, enabling transparent resizing and migration.

3.2 Elastic GPU Compute and GPU-Agnostic Memory Abstraction

GPU-agnostic device memory (for migration and scaling). With native CUDA, device memory allocations and pointers are bound to a specific GPU context, which hinders transparent migration and makes mid-execution resizing costly. Flyt creates a persistent *primary CUDA context* within each GPU virtualization server that owns all the device memory. Importantly, applications never observe physical GPU addresses; all device pointers returned to the application refer to the logical device address managed by Flyt that are translated to the active GPU at runtime. A memory manager associated with the primary context maintains a per-process mapping table that translates application-visible device pointers to physical GPU memory.

Figure 2 shows the address-translation mechanism. Because memory ownership is separated from execution, migration does not require pointer rewriting; only the mapping table changes. This design preserves pointer identity across SM reconfiguration (same GPU) and migration (new GPU). As a result, Flyt is fully transparent to applications, supporting unmodified CUDA binaries without requiring source access, binary rewriting, or guest OS modifications.

Elastic SM provisioning (for runtime resizing). Flyt treats SMs as a dynamically allocatable resource. The applications are executed in

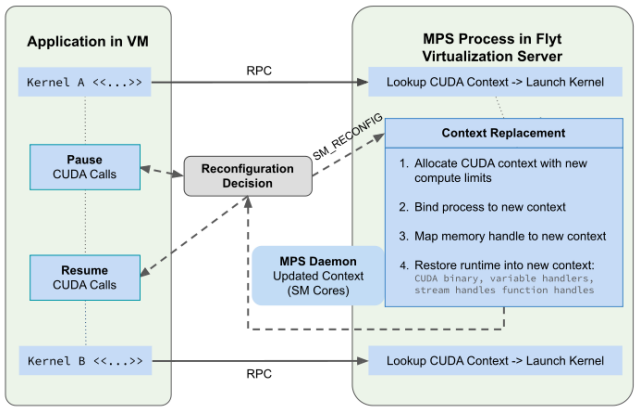


Figure 3: Runtime SM reconfiguration via context replacement. Flyt adjusts SM allocations by replacing execution contexts while preserving application-visible state across reconfiguration.

secondary CUDA contexts that are parameterized by SM execution caps and are linked to the primary context's memory pool. SM changes are realized by *context replacement*: creating a new secondary context with the desired SM cap, restoring runtime state (modules, streams, handles) and resuming execution. This avoids device-memory copies during vertical scaling actions on the same GPU.

Figure 3 shows Flyt's runtime SM reconfiguration. Upon a reconfiguration trigger, the virtualization library pauses CUDA execution at a safe point, creates a new CUDA context with updated SM limits, restores runtime state, and transparently resumes execution after destroying the old context. SM caps are enforced by initializing each virtualization server with `CUDA_MPS_ENABLE_PER_CTX_DEVICE_MULTIPROCESSOR_PARTITIONING=1`, enabling per-context SM limits under NVIDIA MPS. The orchestrator-defined `APP_SM_CORES` is applied as a runtime cap on the SM occupancy. This moves MPS from coarse percentage-based sharing to explicit SM-level quotas, bounding kernel interference, and improving predictability under contention.

3.3 End-to-End Workflow

The operational workflow of Flyts spans system initialization, application on-boarding, steady-state execution, elasticity control, and possible migration across the GPU backend.

Initialization. Each GPU node registers with the orchestrator and begins streaming device telemetry (SM occupancy, memory usage). GPU agents are ready to instantiate GPU virtualization servers on demand.

On-boarding. When an application starts, the virtualization library sends metadata (resource hints, affinity, latency sensitivity) to the orchestrator, which selects a GPU node, assigns initial SM/memory limits, and returns the endpoint; subsequent CUDA calls are redirected to that endpoint. This endpoint indirection decouples application execution from the original binding of the VM's to the GPU devices, allowing per-application placement and remapping across GPU nodes.

Table 3: GPU Node Selection Rules by the Orchestrator.

Policy	Selection Logic	Transport
VM-Dedicated	Uses the GPU currently assigned to the VM.	SHM / TCP
Node Affinity	Selects from user-list if capacity exists.	SHM / TCP
App-Scoped	Selects node with highest available resources.	TCP
Default	Selects first available node with capacity.	TCP

Node selection and affinity. The orchestrator honors the affinity preferences specified by the user using the hierarchical rules in Table 3. For co-located VM–GPU deployments, the data path uses shared memory (SHM) for low-latency, high-bandwidth transfer, while control traffic remains over TCP. For remote placements, both data and the control path use TCP. If the preferred node is unavailable, the orchestrator fall's back to a placement with sufficient capacity to ensure forward progress.

Monitoring and elasticity control. The virtualization library reports kernel launch and execution latency at the VM–GPU boundary, while the GPU server reports SM utilization and memory bandwidth telemetry. The orchestrator fuses these signals to drive elastic control: (i) local vertical scaling by context replacement (adjust SM caps), (ii) priority-aware reallocation when local capacity is tight, and (iii) *targeted migration:* If neither local scaling nor priority-aware reallocation resolves the violation, the orchestrator selects migration as a corrective action; the migration mechanism itself is described in Section 3.4.

Flyt uses a feedback-driven policy over a sliding window of the most recent N tail-latency samples (default $N = 5$). If the windowed latency exceeds the upper (or falls below) threshold, the orchestrator increases (or decreases) the SM cap and enforces a cool-down to prevent oscillations.

3.4 GPU Endpoint Migration

When migration is selected by the orchestration logic described in Section 3.3, Flyt executes transparent GPU endpoint migration as follows. Migration is enabled by a strict separation between memory ownership and execution: applications interact only with Flyt-managed logical device addresses, while physical GPU memory and execution contexts are owned and managed by virtualization servers. This indirection preserves the application-visible pointer identity and the CUDA object semantics across migration. Figure 4 illustrates the migration sequence and control flow. The orchestrator initiates migration by issuing a `PAUSE` command, which quiesces CUDA API submission and drains outstanding asynchronous operations, establishing a consistent execution boundary. The source GPU virtualization server then captures the complete GPU-resident state of the application, including device memory contents, loaded modules, stream and event state, and CUDA handle mappings, and serializes this state to shared storage.

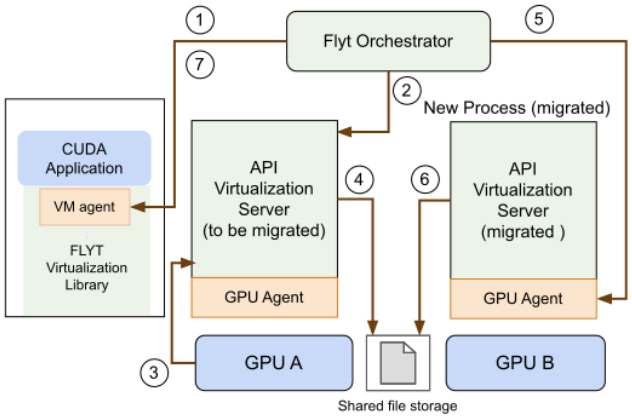


Figure 4: Flyt GPU endpoint migration sequence.

On the destination node, the virtualization server creates a new primary context, allocates GPU memory, reconstructs the address translation table, and restores the captured state to a new secondary execution context configured with the appropriate SM execution caps. Because logical device addresses remain unchanged, migration reduces to rebuilding translation metadata and restoring runtime state rather than rewriting application pointers or modifying application state. CUDA contexts, streams, and handles retain their identity across migration, allowing execution to resume transparently.

Migration is atomic at the application level: execution resumes only after successful restoration on the destination GPU. If migration fails at any stage, execution remains paused on the source GPU, and no partial state is exposed. The cost of migration is bounded by the application's resident GPU memory and associated runtime metadata; host-side application state is not migrated.

4 Evaluation

The evaluation quantifies the overheads and effectiveness of Flyt's two mechanisms—GPU-agnostic memory virtualization and elastic SM execution caps—under dynamic and multi-tenant workloads. We evaluate Flyt in four dimensions: (i) virtualization overhead, (ii) effectiveness of resource management primitives, including vertical scaling, horizontal scaling, and migration, (iii) orchestration behavior under dynamic workloads, and (iv) performance impact on real-world ML inference applications.

4.1 Experimental setup and metrics

Experiments were conducted on a four-node GPUs cluster comprising two Nvidia A5000 (64 SMs, 8192 cores, 24GB) and two Nvidia A4000 GPUs (48 SMs, 6144 cores, 16GB), each node equipped with 64 CPU cores and 64GB RAM. VMs are provisioned with 8 vCPUs and 8GB RAM. The software stack includes NVIDIA 550.54, CUDA 12.4, Ubuntu 20.04, Python 3.12, Torch 2.3. We treat the cluster as SM-homogeneous for scheduling; heterogeneity-aware scheduling is future work. We evaluate the following workloads:

Table 4: Rodinia benchmark application runtime (mean ± std, n=10): native vs. Flyt over TCP and SHM.

Workload	Native (s)	TCP (s)	SHM (s)
Needleman-Wunsch	3.61 ± 0.01	6.34 ± 0.12	5.32 ± 0.02
LU-Decomposition	2.05 ± 0.01	2.53 ± 0.01	2.35 ± 0.01
StreamCluster	56.52 ± 0.11	73.00 ± 0.47	63.26 ± 0.10
SRAD	8.51 ± 0.02	12.86 ± 0.07	9.55 ± 0.03

- **Rodinia benchmark applications:** Standard CUDA applications used as part of the UVMBench suite to exercise GPU compute/memory behavior under our virtualization settings [6].
- **Synthetic workloads:** Matrix multiplication applications with controlled SM and memory demands to test elasticity/migration.
- **ML inference workloads:** TorchServe-based deployment [1, 8] running the following models: DenseNet [9], EfficientNet [21], ResNet [7] and VGG [19].

The key metrics for evaluation include application latency, application throughput, resource utilization, migration overhead, and latency adherence.

4.2 Flyt virtualization overheads

This experiment quantifies the steady-state cost of GPU virtualization under different transport mechanisms. Table 4 reports the mean execution times (n=10) for representative Rodinia workloads under native execution and Flyt using TCP and shared-memory (SHM) transports. Flyt introduces additional overhead relative to native execution, the magnitude of which depends on the characteristics of transport and workload.

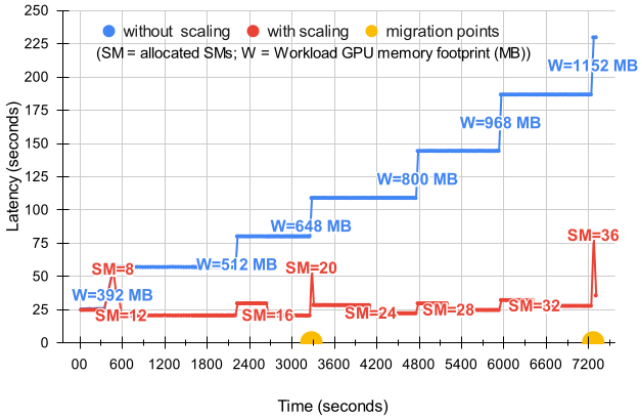
Across all workloads, the TCP-based configuration incurs higher overhead, ranging from 23% (LU-Decomposition) to 51% (SRAD), reflecting RPC serialization and network transport costs on the latency-sensitive CUDA control path. In contrast, SHM reduces overhead for most workloads: LU-Decomposition, StreamCluster, and SRAD remain within 12–15% of native performance.

The differences arise from the characteristics of each workload's control path. LU-Decomposition, StreamCluster, and SRAD are dominated by long-running kernels and issue relatively few CUDA runtime calls; as a result, API interception and transport overheads are amortized over substantial kernel execution time. In contrast, Needleman–Wunsch exhibits substantially higher overhead (~47%) because it iteratively launches many small kernels, with granularity determined by the size of the matrix. This produces frequent kernel launches and CUDA runtime interactions, making performance highly sensitive to API interception and remoting costs. Therefore, the overhead accumulates across iterations, reflecting the control path dominated nature of the workload.

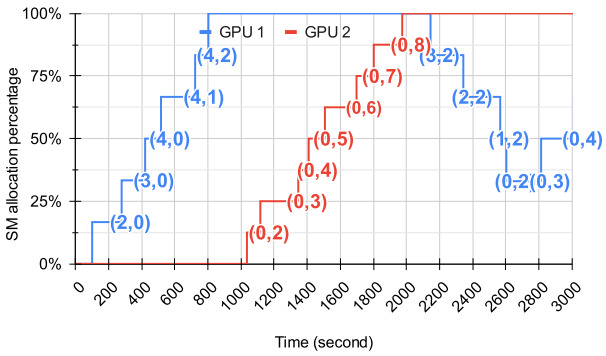
Table 5 breaks down virtualization overheads for data transfer and control operations. With co-located execution, SHM reduces the latency of `cudaMemcpyH2D/D2H` compared to TCP, while control calls (`cudaLaunchKernel`, `cudaGetDeviceCount`) remain dominated by per-call remoting overheads and therefore stay higher than native. These results indicate that transport choice primarily impacts bulk data movement, whereas control path overhead depends on control-call intensity.

Table 5: Data-transfer and control path micro-benchmarks (latency in μs, mean ± std, n = 10).

CUDA API	Native (μs)	TCP (μs)	SHM (μs)
`cudaMemcpyH2D (4MB)`	5.8 ± 0.8	91 ± 12	68 ± 7
`cudaMemcpyD2H (4MB)`	7.1 ± 0.7	105 ± 11	29 ± 5
`cudaLaunchKernel`	2.8 ± 0.9	85 ± 12	62 ± 7
`cudaGetDeviceCount`	0.24 ± 0.02	82 ± 16	58 ± 8



(a) Latency vs. time under static SMs and dynamic SM reallocation; migration triggers and reallocation points annotated.



(b) Per-GPU SM utilization with application admissions and migration; Annotations show the number of active applications as (x, y), where x and y are the counts from VM 1 and VM 2, respectively.

Figure 5: Flyt elasticity: vertical SM resizing and GPU migration under dynamic load.

4.3 Effectiveness of Flyt primitives

Vertical Scaling. We evaluate the impact of runtime SM reallocation on end-to-end application latency using a matrix multiplication workload with increasing input sizes. As shown in Figure 5a, latency over time is measured under static SM allocation and dynamic SM reallocation.

With a static allocation, the latency increases sharply once demand exceeds the initial capacity, and the variance increases with

Table 6: Latency isolation under multi-tenant contention.

Configuration	Avg. MatMul Latency (s)	
	C-App	G-App
C-App only	2.02 ± 0.02	–
C-App + 1 G-App	2.03 ± 0.01	2.05 ± 0.03
C-App + 2 G-Apps	2.02 ± 0.01	4.54 ± 0.02
C-App + 3 G-Apps	2.01 ± 0.02	7.09 ± 0.40
C-App + 4 G-Apps	2.04 ± 0.01	10.29 ± 0.83

C-App denotes a time-sensitive / latency-critical application.
G-App(s) denote best-effort applications.
Reported values show mean latency ± standard deviation.

the input size. With reallocation enabled, latency remains relatively stable as workload scales, with stepwise changes corresponding to increases in allocated SMs. In our experiments, each reallocation incurs a one-time overhead associated with stream context reconfiguration, remaining below 130 ms for workloads with fewer than 60 CUDA streams and increasing to ≈ 180 ms for larger configurations.

Horizontal scaling and migration. This experiment evaluates the cost and effectiveness of redistributing GPU execution across devices once local capacity is exhausted. When local SM capacity is exhausted, Flyt migrates execution across GPUs. In Figure 5a, migration is triggered at $t \sim 3300$s, after which the execution resumes on a GPU with additional SM headroom. The measured migration time (from the detection of sustained deviation to the completion of state restoration) is ∼300ms; each migration induces a transient latency increase of 1.2-1.8×, proportional to resident GPU memory at the time of migration.

Figure 5b shows aggregate provisioning between two GPUs for two VMs in application-scoped mode. VM 1 is capped at 32 SMs and VM 2 at 96 SMs; each application is launched with an initial allocation of 8 SMs. The annotations are shown as (x, y), where x and y denote the number of active applications of VM 1 and VM 2, respectively. VM 1 saturates GPU 1 first; subsequent VM 2 applications consume the remaining capacity on GPU 1 and are placed on GPU 2 once GPU 1 is fully utilized (around $t \approx 800$s), demonstrating that Flyt can extend aggregate capacity beyond a single GPU via admission-time placement, scaling, and migration.

Differentiated Provisioning. Table 6 reports the impact of co-locating multiple instances of a matrix multiplication workload within a single VM under deterministic, application-scoped GPU provisioning. One instance is designated as latency-critical (C-App), while the remaining instances are best-effort (G-App). With one C-App and one G-App, both receive comparable GPU allocations and exhibit similar latency. As additional G-Apps are introduced, their latency increases monotonically due to reduced effective GPU share and contention, while C-App latency remains stable across all configurations, varying by less than 2%. In this experiment, the C-App is assigned a fixed allocation of 8 SMs from a per-VM budget of 16 SMs, and all G-Apps share the remaining capacity. These results demonstrate that Flyt enforces latency isolation for critical workloads while allowing best-effort applications to opportunistically utilize unused GPU resources.

Table 7: Representative orchestration actions under a dynamic multi-application workload.

Timestamp (s)	Action	App count	SM Occupancy (%, SMs)	
			GPU 1	GPU 2
2	Start App	1	8% (4)	0% (0)
13	App ++	2	17% (8)	0% (0)
57	App ++	4	25% (12)	0% (0)
154	Vertical ++[a]	4	42% (20)	19% (12)
398	Vertical ++[a]	4	75% (36)	19% (12)
588	Vertical ++[a]	4	92% (44)	19% (12)
868	Migrate ++[b]	4	83% (40)	25% (16)
1178	App −−	4	8% (4)	44% (28)
2636	Vertical ++[a]	6	58% (28)	56% (36)
2950	SM limit reached[c]	6	92% (44)	56% (36)

[a] Vertical ++ denotes GPU SM allocation increase.
[b] Migrate ++ indicates application migration across GPUs.
[c] SM limit indicates GPU capacity saturation.

Together, these experiments show that elastic SM reallocation accommodates increasing load up to physical saturation. Beyond that point, migration enables capacity expansion on another GPU.

4.4 Orchestration with multiple workloads

We evaluate the dynamic co-execution of multiple latency-sensitive applications across three GPU-enabled VMs. Eight applications (three LU-Decomposition, three SRAD, and two StreamCluster) execute in application-scoped mode with target latency bounds; no affinity is configured and all communication uses TCP.

Table 7 summarizes representative orchestration events, reporting the number of active applications, SM occupancy per-GPU, and the action taken. As applications are admitted (*App++*), the occupancy of SM on GPU 1 increases in discrete steps from 8% (4 SMs) to 92% (44 SMs), reflecting setwise vertical scaling based on preconfigured allocation increments (e.g., 154s, 398s, and 588s). When demand approaches GPU 1 capacity, an application is migrated to GPU 2 (*Migrate++*, 868s), after which the load is distributed across both devices. Subsequent admissions trigger further SM allocation adjustments on both GPUs within configured limits.

Overall, this experiment demonstrates that set-wise SM reallocation and migration are sufficient, in this setting, to incrementally accommodate growing multi-application demand, without centralized device-level repartitioning or VM resizing.

4.5 Control Path Sensitivity

We quantify how CUDA control path intensity affects API-level GPU virtualization overheads, using TorchServe inference workloads with identical SM allocations. As shown in Table 8a, Flyt reduces throughput relative to native execution, with TCP amplifying remoting overheads by one to two orders of magnitude, while SHM substantially mitigates this cost (4–7× higher throughput than TCP). The remaining gap to native indicates that communication and control path overheads dominate performance in these inference settings rather than Flyt SM elasticity or memory indirection mechanisms.

Table 8: ML inference throughput and control path intensity (batch size = 30, 8 SMs per service, single VM).

(a) Inference throughput (images/s).

Application	Native	Flyt (SHM)	Flyt (TCP)
ResNet50	300	20.0	2.0
VGG16	296	55.0	5.2
EfficientNet-B0	305	19.7	1.7
DenseNet121	308	8.1	0.9

(b) Control path intensity across models.

Model	Kernel Launches	H2D Copies	CUDA API Calls
ResNet50	85,331	385	0.79M
VGG16	52,472	161	0.22M
EfficientNet-B0	12,937	998	0.87M
DenseNet121	244,122	1,883	1.85M

Table 8b illustrates model-specific behavior: throughput degradation correlates with control path intensity (kernel launches, H2D copies, and context queries). DenseNet121 shows the highest intensity and degradation of the control path, while VGG16 exhibits a lower intensity and correspondingly higher throughput under Flyt. These results indicate that the optimization of the control path and transport complements elastic SM allocation and migration.

5 Conclusions

This paper presented Flyt, an API-level GPU virtualization solution with elastic SM allocation, nested resource provisioning, and live GPU endpoint migration for latency-sensitive cloud workloads. The key design and implementation aspects of Flyt were using an address indirection mechanism and MPS-based processes for remote API servers. In addition, an orchestrator demonstrated the efficacy of elastic GPU provisioning with Flyt. Evaluation on a multi-GPU cluster using Rodinia benchmarks and TorchServe-based inference workloads quantified virtualization overheads and demonstrated latency control under dynamic load via SM reallocation and migration. Flyt currently relies on CUDA MPS, provides software-based SM partitioning but does not offer isolation or dynamic memory resizing for nested memory usage; therefore, it is best suited for trusted or single-tenant deployments. Future work includes extending migration to workloads with opaque device-side memory, integrating hardware isolation mechanisms such as NVIDIA MIG for stronger isolation with elasticity, and enhancing co-located VM performance by enabling SHM-based control path communication. We believe Flyt demonstrates that software-defined SM-level elasticity is a practical foundation for future GPU sharing systems, bridging the gap between static hardware partitioning and coarse-grained cluster schedulers.

Acknowledgments

This work was supported in part by the IBM AI Horizons Network (AIHN) Research Collaboration with IIT Bombay.

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] Joshua Bakita and James H Anderson. 2023. Hardware compute partitioning on NVIDIA GPUs. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 54–66.

[3] Dejan Bogdanovic, Miroslav Popovic, and Srdjan Usorac. 2021. Analysis of virtio GPU in a containerized environment. In *2021 29th Telecommunications Forum (TELFOR)*. 1–3.

[4] José Duato, Antonio J Pena, Federico Silla, Rafael Mayo, and Enrique S Quintana-Ortí. 2010. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *2010 International Conference on High Performance Computing & Simulation*. 224–231.

[5] Niklas Eiling, Jonas Baude, Stefan Lankes, and Antonello Monti. 2022. Cricket: A virtualization layer for distributed execution of CUDA applications with checkpoint/restart support. *Concurrency and Computation: Practice and Experience* 34, 14 (2022), e6474.

[6] Yongbin Gu, Wenxuan Wu, Yunfan Li, and Lizhong Chen. 2020. Uvmbench: A comprehensive benchmark suite for researching unified virtual memory in gpus. *arXiv preprint arXiv:2007.09822* (2020).

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[8] Sonia Horchidan, Emmanouil Kritharakis, Vasiliki Kalavri, and Paris Carbone. 2022. Evaluating model serving strategies over streaming data. In *Proceedings of the Sixth Workshop on Data Management for End-To-End Machine Learning*. 1–5.

[9] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

[10] 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. arXiv:2409.14447 https://arxiv.org/abs/2409.14447

[11] Yu-Shiang Lin, Chun-Yuan Lin, Che-Rung Lee, and Yeh-Ching Chung. 2019. qcuda: Gpgpu virtualization for high bandwidth efficiency. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 95–102.

[12] Kelvin KW Ng, Henri Maxime Demoulin, and Vincent Liu. 2023. Paella: Low-latency Model Serving with Software-defined GPU Scheduling. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23)*. 595–610.

[13] NVIDIA Corporation. 2024. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[14] NVIDIA Corporation. 2024. Multi-Process Service (MPS) Documentation. https://docs.nvidia.com/deploy/mps/index.html

[15] NVIDIA Corporation. 2024. NVIDIA Multi-Instance GPU (MIG) User Guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/

[16] NVIDIA Corporation. 2025. Virtual GPU Software User Guide - NVIDIA Docs. https://docs.nvidia.com/vgpu/19.0/grid-vgpu-user-guide/index.html

[17] Manos Pavlidakis, Giorgos Vasiliadis, Stelios Mavridis, Anargyros Argyros, Antony Chazapis, and Angelos Bilas. 2024. Guardian: Safe GPU Sharing in Multi-Tenant Environments. arXiv:2401.09290 https://arxiv.org/abs/2401.09290

[18] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. 2011. vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Transactions on computers* 61, 6 (2011), 804–816.

[19] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[20] 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 (EuroSys '24)*. 1075–1092.

[21] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*. 6105–6114.

[22] Sheng Wang, Shiping Chen, and Yumei Shi. 2024. GPARS: Graph predictive algorithm for efficient resource scheduling in heterogeneous GPU clusters. *Future Generation Computer Systems* 152 (2024), 127–137.

[23] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Q Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 595–610.

[24] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 533–548.

[25] Shan Yu, Jiarong Xing, Yifan Qiao, Mingyuan Ma, Yangmin Li, Yang Wang, Shuo Yang, Zhiqiang Xie, Shiyi Cao, Ke Bao, et al. 2025. Prism: Unleashing GPU Sharing for Cost-Efficient Multi-LLM Serving. *arXiv preprint arXiv:2505.04021* (2025).