ISBN: 978-981-94-4198-3
2025 the 15th International Workshop on Computer Science and Engineering (WCSE 2025)
doi: 10.18178/wcse.2025.06.064

Improving GPU-Accelerated Virtual Machines Startup Performance in Cloud Environments

Chia-Chuan Chuang [1 +], Chia-Yen Wu [1]

[1] National Center for High-Performance Computing

Abstract. Cloud-based GPU provisioning is increasingly crucial for AI workloads. However, large-memory GPU VMs often experience lengthy boot times due to the memory reservation requirement. Two principal methodologies are commonly employed to mitigate this issue: huge page memory in host operating system and thread context by the underlying hypervisor. This research investigates the impact of huge pages and thread context on mitigating this startup time bottleneck. Our results quantify the performance gains achieved through these optimizations and demonstrate that configuration of thread context and huge pages can significantly reduce VM startup time by 24% and 80.9%, respectively. This work provides practical guidance for improving the deployment of GPU VMs in cloud environments, contributing to faster application startup.

Keywords: GPU, Passthrough, Virtualization, QEMU, Libvirt

1. Introduction

The growing demand for efficient data processing within GPU-accelerated workflows motivates the increasing use of virtualization for GPU resource provisioning. Existing GPU virtualization technologies, primarily vGPU and passthrough [1], present limitations. vGPU, while enabling shared GPU access, often incurs additional costs through licensing. Passthrough, which offers dedicated GPU access to a single VM, restricts resource sharing and scalability. For cost-sensitive deployments and those requiring near-native GPU performance, passthrough is often the preferred method for enabling GPU access within virtualized environments [2].

A key distinction between standard VMs and those employing GPU passthrough lies in memory management. Passthrough VMs typically impose the constraint of static memory allocation during startup, precluding the dynamic memory allocation capabilities of conventional VMs. In most scenario, people often configure GPU VMs with substantial memory and CPU resources to handle the entire data pipeline, from preprocessing to final analysis. Excessive memory pre-allocation can significantly prolong the VM startup process, leading to frustrating and unpredictable timeout errors for users. This degraded user experience underscores the need for optimized memory management strategies in GPU passthrough environments. Given the performance bottlenecks associated with large memory pre-allocation in GPU passthrough VMs, this work investigates the potential of thread context and huge page mechanisms to mitigate these issues and improve the startup performance of memory-intensive GPU VMs.

2. Related Work

2.1. Virtualization Architecture

QEMU[3], a widely-used open-source type-2 hypervisor, supports full virtualization capabilities. It provides a software-based emulation of a wide range of hardware components, including processors, memory, storage, and peripheral devices, facilitating cross-platform execution of operating systems. QEMU's software-based full virtualization can introduce performance overhead. However, the introduction of Kernel-based Virtual Machine (KVM)[4] allows for near-native performance by leveraging hardware-assisted virtualization. With KVM handling CPU virtualization directly, QEMU's role shifts to emulating remaining peripherals, significantly improving overall performance. The adoption of the QEMU/KVM architecture has

+ Corresponding author. Tel.: +886-4-24620202#856; fax: +886-4-24627373.
E-mail address: jimmy_chuang@narlabs.org.tw.

significantly influenced the evolution and widespread deployment of virtualization technologies within cloud computing. Libvirt [5] is an open-source management tool designed to provide a unified interface for interacting with various virtualization technologies. Libvirt provides an abstraction for simplifying virtual machine management by enabling users and automation tools to perform operations such as creating, starting, stopping, and migrating VMs without requiring in-depth knowledge of the specific underlying hypervisor. By providing a unified abstract interface, libvirt now supports a variety of hypervisors, such as QEMU/KVM, Xen, and VMWare ESXi [6], and facilitates the development of portable virtualization management solutions.

The Fig. 1 illustrates the hierarchy relationship between QEMU and Libvirt, in which Libvirt serves as a management layer atop QEMU. Moreover, cloud manager systems like OpenStack commonly utilize libvirt to abstract the complexities of underlying hypervisors, thereby simplifying and standardizing the system architecture. Fig. 1 also depicts the different layers of abstraction available for VM provisioning, ranging from direct interaction with the cloud management system to lower-level control via the libvirt API or direct execution of hypervisor commands. The hierarchical structure simplifies VM management, shielding users from the complexities of direct hypervisor interaction while leveraging its underlying virtualization capabilities.

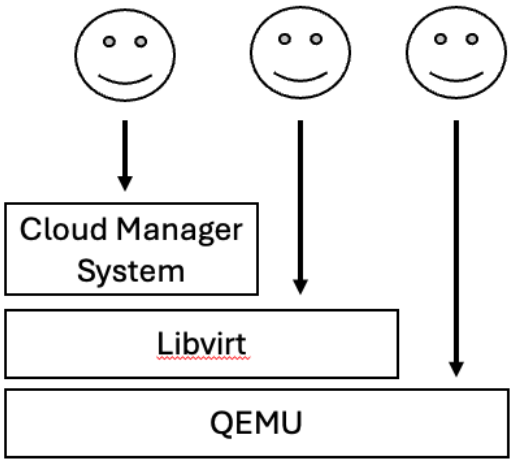

Fig. 1: Virtualization Hierarchy Architecture.

2.2. I/O Virtualization

I/O virtualization plays a crucial role in achieving efficient and flexible resource management within virtualized environments. Several approaches exist for implementing I/O virtualization, each with its own trade-offs regarding performance, complexity, and flexibility. The most common techniques include:

1. Full Virtualization: The hypervisor emulates all I/O operations in software. This provides high compatibility but can introduce significant low performance.
2. Paravirtualization: This relies on modified guest operating systems (OS) with specialized drivers that communicate directly with the hypervisor. Paravirtualization improves performance compared to full virtualization but requires modifications to the guest OS, limiting compatibility.
3. Single Root I/O Virtualization (SR-IOV): This allows a single physical I/O device to be presented as multiple virtual functions (VFs) to different VMs. However, SR-IOV support depends on specific hardware and may not be available for all devices.
4. Passthrough: This assigns a physical I/O device directly to a single VM, providing near-native performance and bypassing the hypervisor's I/O virtualization layer. This offers the highest performance but limits resource sharing as the device is dedicated to a single VM.

Balancing the competing demands of cost, performance, and compatibility, passthrough often emerges as the preferred method for GPU virtualization. This choice is motivated by the requirement for maximizing GPU performance within the virtualized environment, as passthrough offers the lowest overhead and allows the VM to access the full capabilities of the physical GPU. While this approach limits resource sharing, it is deemed the most suitable for our specific workload and performance requirements.

When utilizing GPU passthrough, the underlying IOMMU (Input/Output Memory Management Unit) facilitates device assignment by requiring contiguous physical memory regions to map the GPU's memory space into the guest VM's address space. This process is essential for enabling direct access to the GPU by the guest VM. Unlike standard VMs where memory can be dynamically allocated during runtime, the requirement for pre-allocated, contiguous memory regions for GPU passthrough can significantly increase boot times, particularly for VMs configured with large memory capacities.

3. Impact of Thread Context and Huge Pages

Huge pages and thread context are two mechanisms that can contribute to improved boot times for VM configured with substantial amounts of memory.

Huge pages are a memory management feature provided by the operating system. With standard 4KB pages, managing large memory regions requires extensive page tables, increasing the overhead for address translation. Huge pages increase the page size used for memory allocation, reducing the overhead associated with page table lookups. Thread Context is a memory allocation enhancement mechanism built in hypervisor, not in operating system. This feature introduced in QEMU version 7.2.0 that enhances the efficiency of memory preallocation for virtual machines, particularly those with large memory configurations. Thread Context enables parallel initialization of memory regions, further accelerating the preallocation process. This is particularly beneficial for VMs configured with substantial amounts of memory.

3.1. Experiment Results

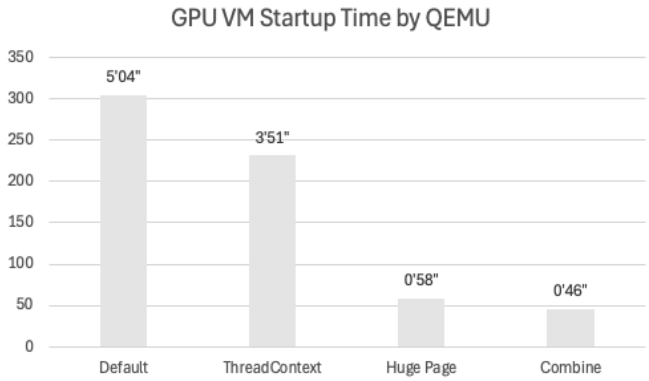

Fig. 2: Startup time for VM launched by QEMU

This section presents the experimental results on VM boot times, exploring the effects of thread context and huge pages on startup performance. The test VM consists of a system with 48 CPU cores, 512GB of memory, and a dedicated GPU, and the test VM is launched by QEMU directly.

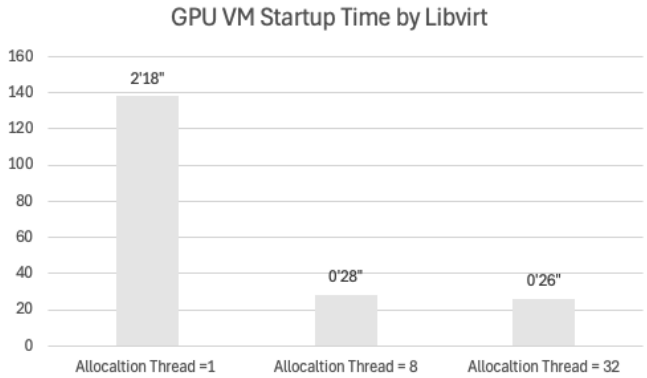

Fig. 3: Startup time for VM launched by Libvirt

Fig. 2 provides a comparsion of VM creation times within the QEMU environment, examining the effects of enabling and disabling both huge pages and thread context optimizations. The base line of a GPU VM boot time is 5 minutes and 4 seconds. Enabling thread context alone reduces the boot time to 3 minutes

and 51 seconds, a significant improvement about 24%. Employing huge pages alone dramatically reduced the boot time to 58 seconds, an improvement of 80.9% compared to the baseline configuration. Combining both huge page and thread context leads to the fastest boot time of 46 seconds, but this is only slightly faster than using huge pages alone. The data suggest that huge pages are a more portable optimization for accelerating GPU VM startup performance. While thread context benefits are tied to QEMU, the advantages of huge pages extend to other hypervisors, offering broader applicability.

Libvirt provides an allocation thread feature, similar in function to thread context. Fig. 3 illustrates the performance impact of various allocation thread configurations while holding the huge page setting constant and enabled. When Libvirt employs only a single thread, VM startup incurs a noticeably higher overhead. Increasing the number of threads progressively reduces boot time, although this improvement does not scale indefinitely. In our experiments, performance gains plateaued at eight threads.

4. Conclusions

This paper investigated the impact of huge pages and thread context on the boot times of GPU passthrough VMs. Our experimental results demonstrate that both mechanisms independently reduce the time required for memory reservation during VM startup, with huge pages exhibiting a more pronounced effect. While both techniques can be employed separately, leveraging huge pages alone yielded an 80.9% improvement in boot time. Thread context alone resulted in a 24% improvement. When implemented together, these optimizations achieved an 84.8% reduction in boot time. Furthermore, the inherent portability of huge pages, stemming from implementation as an operating system feature, contrasts with the hypervisor-specific nature of thread context, which are tied to the QEMU. Consequently, huge pages offer greater flexibility and broader applicability across diverse virtualization environments employing various hypervisors.

5. References

[1] C. -T. Yang, H. -Y. Wang, W. -S. Ou, Y. -T. Liu and C. -H. Hsu, On implementation of GPU virtualization using PCI pass-through. 4th IEEE International Conference on Cloud Computing Technology and Science Proceedings, 2012, pp. 711-716.

[2] J. Prades and F. Silla, A Live Demo for Showing the Benefits of Applying the Remote GPU Virtualization Technique to Cloud Computing. 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2017, pp. 735-738.

[3] F. Bellard, QEMU, a fast and portable dynamic translator. Proceedings of the annual conference on USENIX Annual Technical Conference, Anaheim, CA, Apr. 2005.

[4] A. Qumranet, Y. Qumranet, D. Qumranet, U. Qumranet, and A. Liguori, KVM: The Linux virtual machine monitor, Proc. Linux Symp., vol. 15, Jan. 2007.

[5] Libvirt: The virtualization API. https://libvirt.org/ (accessed Jan., 2025).

[6] J. P. Walters et al., GPU Passthrough Performance: A Comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL Applications. 2014 IEEE 7th International Conference on Cloud Computing, 2014, pp. 636-643.