

Confidential AI Performance with Intel® Trust Domain Extensions (Intel® TDX) and NVIDIA* H100 GPU: System Level Analysis

By Shiny Sebastian

April 2026

Contents

1.0 Key Takeaways	4
2.0 Introduction	5
3.0 Performance Framework	6
4.0 Tier-1: Architectural Performance Overhead Sources	7
4.1 Intel CPU Confidential Computing Factors	7
4.1.1 Memory Encryption	7
4.1.2 Inter-socket Encryption	7
4.2 Intel TDX-related Factors	8
4.2.1 VMM Exit Duration Increase Due to Intel TDX Transitions	8
4.2.2 VMM Exit reasons	9
4.2.3 NVIDIA H100 GPU CC related overheads	13
5.0 Tier-2: GPU Runtime Execution Effects	18
5.1 Single-Kernel/Graph Queuing Behavior	18
5.2 Multi-Kernel and Multi-Graph Queuing Behavior	19
6.0 Tier-3: System-level Performance Characterization	22
7.0 Performance Evaluation Studies	23
7.1 GPU-CPU Data Path Characterization with the NVBandwidth Tool	23
7.1.1 Confidential Computing Performance Overheads	24
7.1.2 Understanding the Workload	24
7.1.3 Tier-1 Architectural Performance Overhead Reasons	25
7.1.4 Tier 2: GPU Runtime Execution Effects	26
7.1.5 Tier-3 System-level Performance Characteristics	29
7.2 Model Capacity Scaling Analysis with vLLM	29
7.2.1 Understanding the vLLM Serving Benchmark	29
7.2.2 Confidential Computing Performance Overheads	30
7.2.3 Tier-1 - Architectural Performance Overhead Sources	33
7.2.4 Tier-2: GPU Run time Execution Effects	35
7.2.5 Tier-3 System level performance characteristics	37
7.3 Concurrency and Sequence-length Scaling Analysis with vLLM	39
7.3.1 Concurrency scaling	40
7.3.2 Input Length scaling	42
7.3.3 Output Length scaling	45

8.0 Optimization and Mitigation Summary	49
9.0 Summary	52
10.0 References	53
11.0 Notices and Disclaimers	54
11.1 Configurations	54
11.2 Workload and Tracing Commands	55
11.2.1 Model size scaling:	55
11.2.2 Concurrency+ Input + Output scaling	55
11.2.3 Nsight Systems tracing	55
11.2.4 NVBandwidth CE test	56
11.2.5 Enable CC mode in GPU	56
11.2.6 Disable CC mode in GPU	56
11.2.7 Enable CC debug mode in GPU (for Nsight Systems profiling)	56

1.0 Key Takeaways

As AI workloads increasingly process sensitive data at scale, confidential computing (CC) has emerged as a critical security primitive — yet the performance implications of CC on GPU-accelerated inference have yet to be comprehensively characterized. This paper shows that the performance impact of CC is workload-dependent, predictable, and modest for large-scale inference. We present an analytical framework and experimental results that characterize this overhead across model scales and deployment configurations.

Our analysis reveals that the performance impact of confidential computing is primarily determined by the relative proportion of workload time spent in GPU kernel execution versus CPU–GPU control and data-exchange operations. When runtime is dominated by GPU kernel execution, the overhead associated with encrypted communication and trust domain transitions becomes largely amortized. In contrast, workloads that involve shorter compute intervals and more frequent host–device interactions are more sensitive to this overhead.

This trend is reflected in our experimental results. Smaller models ($\approx 1\text{B}$ parameters), which incur relatively more CPU–GPU interaction, experience approximately 10–12% performance degradation under confidential computing conditions. In contrast, larger models (20B–70B parameters), whose execution is dominated by sustained GPU kernel execution, exhibit a much smaller (2–3%) performance impact. Importantly, GPU graph/kernel execution itself remains unaffected, since computation performed within the GPU, using the streaming multiprocessors (SMs) and the on-device memory hierarchy, occurs within the trusted computing boundary and does not traverse encrypted PCIe links. Notably, concurrency scaling and input/output batching further reduce CC overhead — in the 1B model case, these optimizations bring overhead to near parity with non-confidential execution.

Taken together, these results indicate that confidential computing can be deployed for modern GPU-based AI workloads with modest and predictable overhead, particularly for large models where GPU execution dominates runtime, making secure AI deployment both practical and economically viable.

2.0 Introduction

Confidential computing enables protection of data in use by isolating workloads within hardware-enforced trusted execution environments. Technologies such as Intel® Trust Domain Extensions (Intel® TDX) extend this protection to virtualized environments, providing isolation designed to protect sensitive workloads from the host operating system, hypervisor, and other untrusted software. As GPU-accelerated AI workloads increasingly move to shared and cloud environments, understanding the performance implications of running these workloads inside confidential virtual machines becomes critically important.

Large Language Model (LLM) inference workloads are particularly relevant in this context. They rely heavily on high-throughput GPU execution, frequent CPU–GPU communication, and large memory transfers. Confidential computing strengthens security by introducing architectural mechanisms such as encrypted memory, protected device communication, and additional transition boundaries. However, these mechanisms can introduce latency and bandwidth overhead that may affect end-to-end inference performance.

This paper presents a structured performance analysis of GPU-based AI inference workloads running under Intel® TDX with NVIDIA* H100 GPUs^{Ref: 1}. Using both microbenchmarks (NVBandwidth) and real-world serving workloads (vLLM), we analyze how confidential computing overhead originates at the architectural level, propagates through runtime behavior, and ultimately manifests in workload-level metrics such as throughput and latency. To systematically study these effects, we introduce a three-tier analytical framework that separates root causes from runtime effects and workload-level impact.

Our goal is not only to quantify performance impact but also to identify the conditions under which confidential computing overhead is amplified or diminished. The results demonstrate that while smaller or launch-intensive workloads may observe measurable performance degradation with confidential computing, large-scale, compute-dominated LLM inference workloads experience modest and predictable overhead. These findings provide practical guidance for deploying secure AI workloads in confidential environments at scale.

Disclaimer: This work focuses on system-level performance behavior observable through profiling tools, architectural documentation, and runtime measurements. While NVIDIA GPU firmware and internal security processors play an important role in confidential execution, detailed microarchitectural analysis of proprietary hardware and firmware components is beyond the scope of this paper. Instead, we treat the GPU security mechanisms as a protected subsystem and analyze their performance impact through externally measurable effects on CPU–GPU communication, kernel launches, and memory transfers.

Please note also that the experimental setup uses KVM (Kernel-based Virtual Machine) as the VMM; consequently, function references and implementation details are specific to this environment.

3.0 Performance Framework

To systematically analyze confidential computing overhead in GPU-accelerated AI workloads, we introduce a three-tier analytical framework (as shown in Figure 1) that separates architectural root causes from runtime manifestations and workload-level impact. While the broader concept described in Tier-2: GPU Runtime Execution Effects has been explored in prior work^{Ref: 2}, our framework and conclusions are independently derived from empirical observation.

- Tier-1: Architectural Performance Overhead Sources represents the fundamental mechanisms introduced by confidential computing. These include encrypted CPU–GPU communication, Intel TDX transition latency, private-to-shared page conversions, and GPU security processing. Tier-1 factors operate at the hardware and virtualization boundary and form the primary sources of overhead.
- Tier-2: GPU Runtime Execution Effects captures how Tier-1 perturbations influence observable execution behavior. These effects include increased GPU kernel launch latency, extended Host to Device (H2D) and Device to Host (D2H) transfer durations, amplified synchronization time, and queueing delays. Tier-2 represents the performance consequences within the CUDA* runtime and GPU execution pipeline.
- Tier-3: System-level Performance Characterization describes how Tier-2 effects manifest in end-to-end workload metrics such as throughput, Time to First Token (TTFT), Time per Output Token (TPOT), and Inter-Token Latency (ITL). At this level, the specific workload configuration (model size, concurrency, input/output length) determines whether architectural overhead is amplified or amortized.

This layered separation allows us to distinguish between fixed architectural costs and workload-dependent performance behavior. Throughout the paper, we apply this framework to microbenchmarks and large-scale LLM serving workloads to identify both root causes and mitigation strategies.

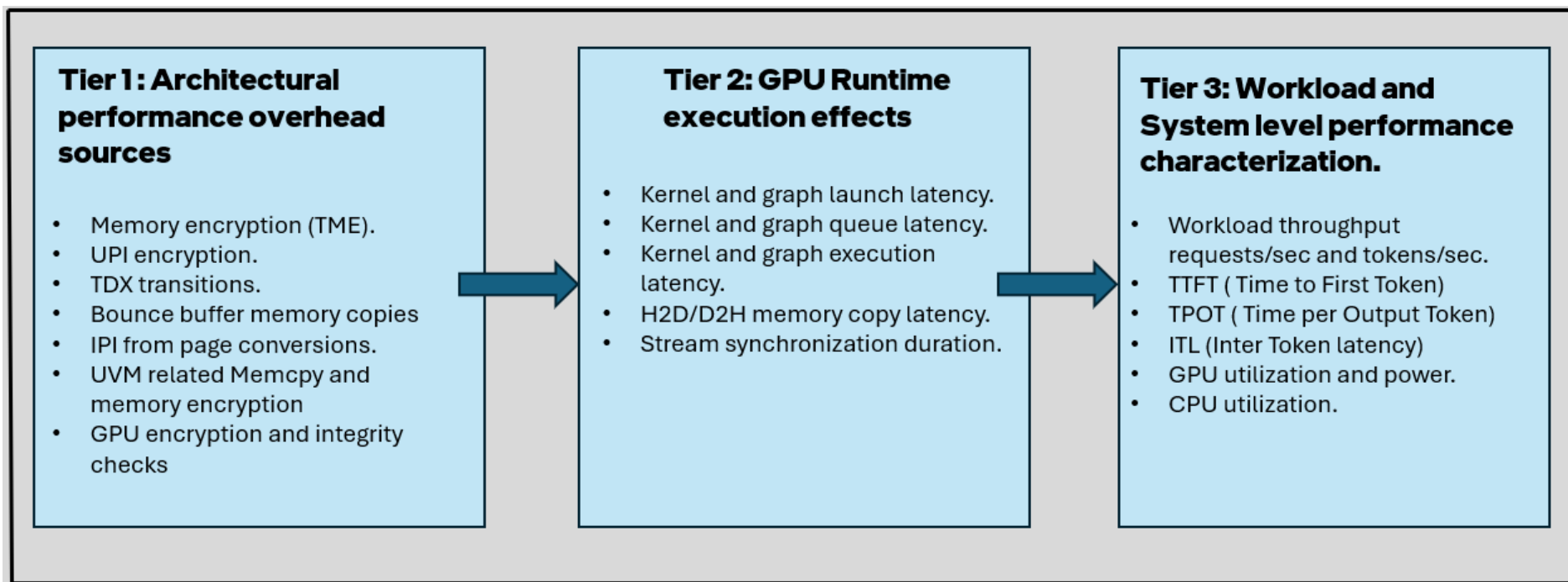


Figure 1: Performance Framework Tiers

4.0 Tier-1: Architectural Performance Overhead Sources

Factors affecting confidential computing performance for AI workloads with Intel TDX and NVIDIA H100 can generally be grouped into three categories: Intel CPU confidential computing factors, Intel TDX-related factors (Intel® TDX Documentation)^{Ref: 3}, and NVIDIA GPU CC-related factors.

4.1 Intel CPU Confidential Computing Factors

4.1.1 Memory Encryption

Both Intel® Software Guard Extensions (Intel® SGX) and Intel TDX are built on top of Intel® Total Memory Encryption – Multi-Key (Intel® TME-MK) technology, which provides multiple hardware-generated ephemeral keys that remain confined to hardware and are not exposed to software or external interfaces. To support this feature, the processor has an AES-XTS encryption engine in the memory subsystem that encrypts data writes and decrypts data reads, to and from DRAM outside the SoC (System on a Chip). A read operation can experience additional latency depending on the DRAM speed, while write operation overhead is less apparent to the applications due to the write-back nature of the CPU cache. For CPU-intensive workloads, Intel TME/TME-MK can cause ~2-3% performance overhead, which can vary depending on workload characteristics and system configuration^{Ref: 4}.

To minimize the impact of memory encryption on non-confidential guests, the SoC supports a feature called Intel® Total Memory Encryption bypass (Intel® TME-bypass), where the memory accesses by the VMM and any other non-trust domain (TD) software bypasses the memory encryption flows in the memory subsystem, thereby reducing the latency impact on read and write operations. Performance impact of Intel TME-bypass is ~1%, which can also vary depending on the nature of the workload and system configuration^{Ref: 3}.

This work focuses on the incremental impact of Intel TDX and NVIDIA Confidential Computing, with Intel TME enabled in both baseline and confidential configurations. Hence, later, when we refer to non-confidential workloads, we will assume that Intel TME is enabled.

On 6th Generation Intel® Xeon® Scalable server processors code named Granite Rapids and earlier processors, switching to faster DIMMs can help reduce the performance overheads due to Intel TME/TME-MK and Intel TME-bypass as compared to non-confidential, TME-disabled environments.

4.1.2 Inter-socket Encryption

Remote memory accesses occur when a CPU accesses memory attached to another socket. In multi-socket systems, such accesses go over the Intel® Ultra Path Interconnect (Intel® UPI) connecting the sockets. When Intel SGX and Intel TDX are enabled on the platform, such remote memory accesses can be encrypted over Intel UPI and thus can experience additional latencies.

To mitigate these extra latencies and in some cases bandwidth limitations in multisocket environments, we recommend an optimal platform and software configuration that ensures all the memory accesses are issued to the memory local to the processor or I/O device.

4.2 Intel TDX-related Factors

In a virtualized environment, VMM exits are an important factor that affects workload performance. During GPU inference in confidential computing environments, these overheads can increase due to two primary reasons: 1) Increase in overall VMM exit/entry durations due to Intel TDX transitions as discussed in the following sub-section 2) Additional VMM exits due to page conversions. Bounce buffer related memory copies due to I/O like disk/network are not observed during inference.

4.2.1 VMM Exit Duration Increase Due to Intel TDX Transitions

Intel TDX introduces a new CPU mode, known as the Secure Arbitration Mode (SEAM)^{Ref: 5} [Reference 1], which is used to support trust domains (TDs). In SEAM, a new software module, known as the Intel TDX Module, runs to support the Intel TDX architecture. Similar to Virtual Machine Extensions (VMX), where transitions occur between the VMM and legacy guests, Intel TDX introduces transitions, TD-exit and TD-entry, between the Intel TDX Module and TDs. However, the CPU state of a TD (that is, general purpose registers, control registers, and MSRs) may contain sensitive data. To prevent any inference of the TD's CPU state, the Intel TDX module saves the TD's CPU state and scrubs it before passing control to the VMM. Similarly, when the TD resumes, the state of the TD is restored from the state-save area.

As shown in Figure 2, with Intel TDX, the VM entry path now additionally includes SEAMCALL—an instruction used by the host VMM to invoke host-side Intel TDX interface functions—and the VM exit path includes SEAMRET—an instruction executed by the Intel TDX module to transfer control back to the VMM. These additional steps, along with other precautions taken by the Intel TDX module, cause Intel TDX transitions to take a longer time than VMX transitions. As a result, workloads inside a TD may cause extra overhead, particularly those that incur frequent Intel TDX transitions. A typical Intel TDX transition would include TDCALL + Module execution + SEAMRET + VMM processing + SEAMCALL + Module execution + VM Resume.

To reduce the performance effects of Intel TDX transitions, software can be tuned to reduce the Intel TDX transition frequency, for example, by using polling mode I/O threads or by reducing timer frequency.

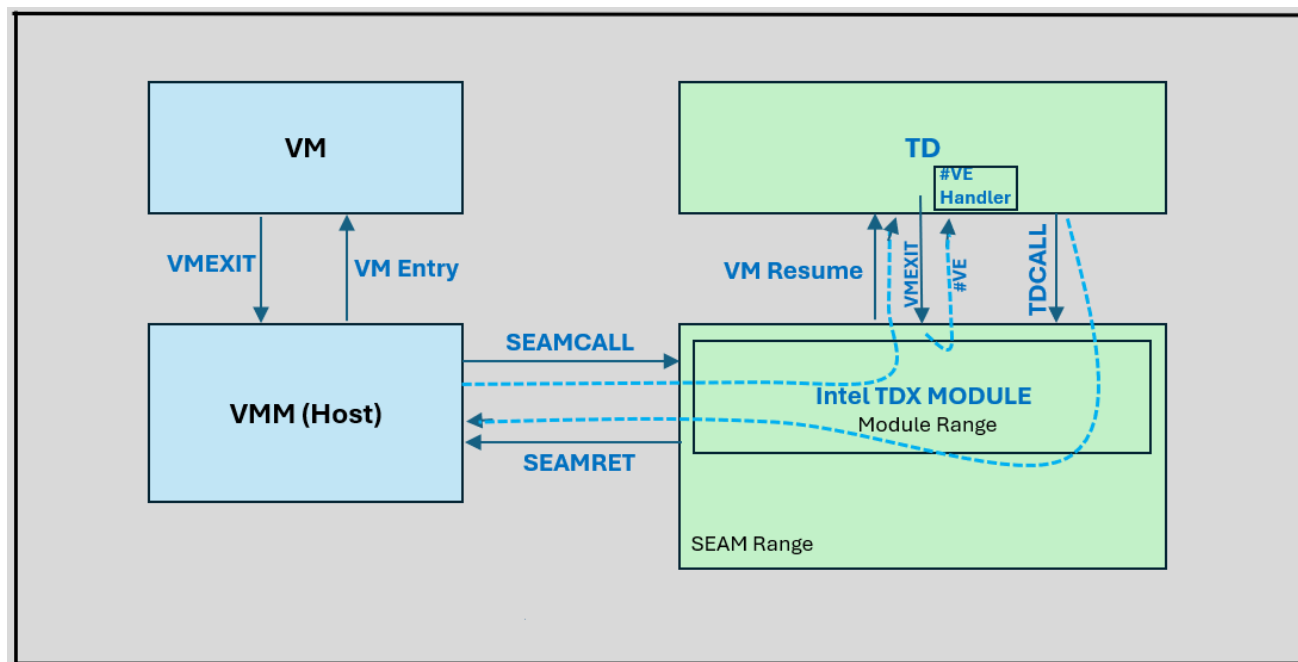


Figure 2: Intel TDX transitions

4.2.2 VMM Exit reasons

During GPU inference, there are primarily two causes of VMM exits: the first is related to page conversions, and the second is due to timer interrupts. These causes are summarized in Figure 3. As per our observations, the rate of VMM exits due to page conversions is significantly lower than the rate of timer interrupts. Let's examine both causes:

4.2.2.1 Page conversions

When the CUDA runtime establishes a device context (for example, during device initialization or first CUDA usage), it transitions into the NVIDIA kernel driver ([nvidia.co](https://nvidia.com)). The driver then initializes GPU-CPU communication mechanisms, including command submission infrastructure (such as push buffers) and allocates kernel-side data buffers used for direct memory access (DMA) transfers (via mechanisms such as `dma_alloc_attrs`).

In non-Intel TDX Connect systems (like bounce buffer based confidential computing environments), the GPU doesn't have direct access to the confidential guest's private memory. Therefore, before the GPU can perform DMA operations, the TD guest must first convert private encrypted pages to shared decrypted pages via `set_memory_decrypted`. This conversion triggers the following chain of events:

These page conversions trigger Translation Lookaside Buffer (TLB) invalidations on both the guest side — covering Guest Virtual Address (GVA) to Guest Physical Address (GPA) mappings and GVA to Host Physical Address (HPA) mappings — and the host side, where GPA to HPA mappings are invalidated within the Secure Extended Page Tables (SEPT).

4.2.2.1.1 Guest side TLB flush

1. In the guest, the dma-direct allocation path may invoke `set_memory_decrypted()`, indicating that the allocated DMA buffer is being converted from guest-encrypted/private memory semantics to shared/unencrypted memory for device access.
2. This function in turn invokes `cpa_flush` (Change Page Attribute Flush function), which initiates TLB invalidation.
3. TLB shutdown is implemented via inter-processor interrupts (IPIs), issued through `native_send_call_func_ipi` and `x2apic_send_IPI_allbutself`.
4. These IPIs are sent via writes to the x2APIC ICR (MSR 0x830). In Intel TDX environments, such MSR writes are intercepted by the Intel TDX module, resulting in VM exit to the Intel TDX Module. The Intel TDX Module then injects a #VE and VM-enters into the guest TD. Then the guest TD sends a TDVMMCALL requesting the host VMM to send an IPI.
5. The VMM emulates the IPI delivery and injects interrupts into target vCPUs. Since these are posted interrupts, they do not cause a TD exit.
6. Upon receiving the interrupts, target vCPUs execute function callback handlers to perform TLB shutdown and invalidate stale TLB translations.

4.2.2.1.2 Host side TLB flush

Additionally, the host may perform its own SEPT (GPA to HPA) flush aka SEPT translation invalidation. This is detailed in Intel TDX module base spec^{Ref: 5} under "Dynamic Page Removal (Private to Shared Conversion)"

After the ICR register write emulation mentioned above, the VMM does a TLB tracking sequence followed by issuing IPIs to other vCPUs of the TD guest:

1. The VMM first blocks the GPA memory range with `TDH.MEM.RANGE.BLOCK`.
2. Next, the VMM increments its TD (global) epoch via the `TDX.MEM.TRACK` instruction.
3. Then it issues IPI to all vCPUs of the TD to flush its SEPT entries.
4. If the vCPU is already running, it takes an exit. These IPIs show up as external interrupts for the vCPUs. When the vCPU does a VM entry into a guest TD, the Intel TDX module flushes the TLB and partial translations, increases its per vCPU epoch, and resumes operation.
5. If the vCPU was not running, then during the next VM-entry of the TD vCPU, it flushes its SEPT and increases its per vCPU epoch.
6. Once all vCPUs are flushed, then the page is removed from private pool with `TDH.MEM.PAGE.REMOVE`.

The corresponding KVM function call sequence is as follows:

`tdx_sept_remove_private_spte -> tdx_track -> kvm_make_all_cpus_request(kvm, KVM_REQ_OUTSIDE_GUEST_MODE)`, where:

- `tdx_sept_remove_private_spte`: removes the target page entry from SEPT.
- `tdx_track`: increases TD's TLB epoch.
- `kvm_make_all_cpus_request`: broadcasts IPIs to all vCPUs, forcing all TD vCPUs to exit guest mode.

These operations contribute to confidential computing overhead for the following reasons:

- VMM exits from APIC ICR MSR writes and external interrupts.
- TLB tracking sequence overhead as described above.
- Each VMM exit incurs longer transition latency due to Intel TDX (TD-exit/TD-entry alongside SEAM execution).
- TLB shutdowns and cache flushes are inherently expensive operations.

Although the guest kernel in our setup allocates up to 1 GB of SWIOTLB buffers, it follows a dynamic decryption policy that triggers page conversions and consequently IPIs at workload runtime, rather than at TD boot time with GPU passthrough.

4.2.2.1.3 Recommendations to potentially optimize these overheads

- Reduce runtime page-conversion overhead by pre-establishing a pool of DMA-accessible shared pages so that `set_memory_decrypted()` is not invoked in the critical path of workload execution. In the traced path, DMA buffers are allocated through `dma_direct_alloc()` and converted on demand, which triggers `cpa_flush`, x2APIC-ICR-related VM exits, and guest TLB shutdowns. A promising direction is to evaluate whether a boot-time or initialization-time shared DMA buffer pool can amortize these conversions outside the workload's steady-state execution.
- Potentially limit TLB shutdown fanout by constraining CPU affinity: TLB shutdowns target CPUs that may have cached translations for a given memory mapping. You could reduce the number of CPUs affected by confining the workload to a smaller CPU mask. For example, pin workload threads to CPUs 5–7 (using `taskset`, `cgroups cpuset`, or `systemd` CPU affinity) and avoid running other threads of the same process on different CPUs.
- Enable IPI virtualization to reduce performance overhead of VMM exits from ICR register writes once its supported ^{Ref: 6}.
- Leverage Intel TDX Connect for passthrough devices: Intel TDX Connect is expected to reduce or eliminate shared-buffer-related page conversions in device passthrough scenarios by allowing trusted device access to TD memory, thereby eliminating private-to-shared page transition overhead.

4.2.2.2 APIC Timer interrupts

Both the host and the guest of the system under test have TSC (deadline) one-shot timer in NO_HZ_IDLE mode. In NO_HZ_IDLE mode, the timer doesn't tick unless there is something scheduled to be run on the processor. It works by using TSC one-shot mode, where the kernel calculates when the next event is needed and programs the LAPIC next deadline MSR via the `lapic_next_deadline` function, which fires the timer once and then stays silent.

4.2.2.2.1 Recommendations to potentially reduce these overheads

Please refer to KVM and OS/kernel documentation to find ways to reduce APIC timer related exits ^{Ref: 7}. For instance, reducing timer frequency in host and guest kernel configurations and restricting the set of CPUs on which the timer can tick via the `nohz_full` kernel boot time parameters on both guest and host.

Also, reduce scheduler-related IPIs for rescheduling using `nohz_full`: For example, you can isolate cores using `nohz_full=5-7` and `rcu_nocbs=5-7`. This reduces scheduler tick interrupts and RCU callback activity on those cores, minimizing rescheduling-related IPIs. (Note: this does not directly limit TLB shutdowns but reduces interrupt noise and improves determinism.)

Hence KVM exit types with this workload can be divided into two categories, and exit rates estimates are quoted from the *nvbandwidth* tool mentioned in the next section to help understand their relative significance:

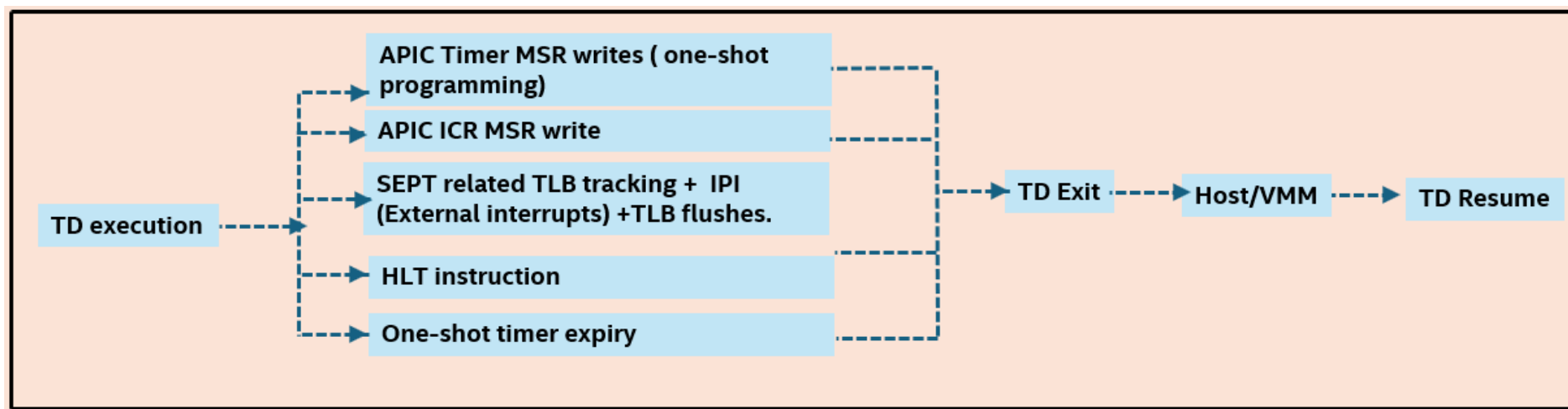


Figure 3: KVM exit reasons with Intel TDX + NVIDIA CC during Inference

4.2.2.2 Synchronous exits

- **Timer Setup:** A timer event is programmed by the guest kernel via the `IA32_TSC_DEADLINE_MSR` to trigger a future interrupt. These MSR writes cause VMM exits, observed as ~1400 exits/sec on the host.
- **IPI send:** When a vCPU initiates a page conversion it writes to the x2APIC Interrupt Command Register (ICR, MSR 0x830) as explained before. These MSR writes are intercepted by Intel TDX module, resulting in VMM exits, seen as ~150 exits/sec on the host
- **HLT:** When a vCPU becomes idle (that is, no runnable tasks are available), if the vCPU has been idle for long, it executes a HLT instruction and remains halted until woken by a timer or IPI interrupt. This is observed as `kvm_exit (HLT)`.

4.2.2.3 Asynchronous exits

This corresponds to class of exits caused by interrupts and preemption timer.

- **IPI Arrival:** Host side TLB invalidations can trigger IPI related VMM exits (`EXTERNAL_INTERRUPT`) on all vCPUs of the TD. Guest side TLB invalidations (not VMM exits) cause a corresponding call-function interrupt handling within the guest on respective target vCPUs.

- Guest timer expiration: When the physical TSC reaches the deadline, a timer interrupt is generated. If the vCPU is running, this may trigger an external-interrupt VM exit, allowing KVM to inject the interrupt into the guest. This is observed as 1400 Timer interrupts per second as measured from within the guest.
- Host timer expiration: Host-side timer interrupts (that is local APIC timer ticks) can preempt a running guest vCPU, forcing a VMM exit so the hypervisor can service the scheduler. After handling the interrupt on the host, KVM/VMX resumes the guest.

Math: 1,400 (Guest Timer expiration) + 1,100 (IPI arrival and Host Timer expiration) $\approx$ 2,500 total External Interrupt Exits.

4.2.2.3 I/O virtualization in Intel TDX

TD's memory contents are protected from untrusted software, BIOS, or firmware unless the TD explicitly shares a memory range with the untrusted entities, for example, to perform I/O operations or to invoke services of the VMM. That shared memory range is called TD shared memory, in contrast to TD private memory, which is not shared with untrusted software. When the TD exchanges data with emulated devices (like the virtio device for KVM and vmbus for Hyper-V*) via TD shared memory, it is the responsibility of the TD to copy data from the shared memory used by devices into the TD's private memory used by workloads running inside the TD and vice versa. This extra memory copy may cause some overhead to I/O intensive workloads that involve a large amount of data exchange between devices. To reduce the effect of memory copy overhead and to improve overall I/O performance, Intel will introduce Intel® Trust Domain Extensions Connect^{Ref: 8} in future Intel CPU generations.

4.2.3 NVIDIA H100 GPU CC related overheads

Since we do not have visibility into the deeper internal operations of NVIDIA's hardware and software stacks, this section leverages insights and informed speculations from external publications.^{Ref: 9,10}

4.2.3.1 Unified Virtual Memory (UVM) and GPU side security mechanism

Unified Virtual Memory (UVM) is a CUDA memory model that provides a single virtual address space that is shared between the CPU and GPU, allowing both processors to access the same pointers without explicit data movement calls. Instead of requiring programmers to manually allocate separate host and device buffers and manage `cudaMemcpy` operations, UVM lets the runtime transparently manage where data resides and when it moves. In essence, the UVM driver `uvm-nvidia.ko` serves both as a runtime fault handling engine and the memory manager for managed memory allocations.

In a non-confidential guest, when the GPU accesses a non-resident page and raises a page fault, the GPU driver (`nvidia.ko`) services the fault by initiating a transfer of the page from its current location in host memory (DRAM) to device memory (HBM). The driver programs the GPU's DMA engine (Copy Engine) to perform this transfer, after which execution resumes. Consequently, the CPU-GPU interconnect (for example, PCIe) lies on the critical path for data transfer performance, rather than CPU execution itself.

However, in a confidential VM, the GPU (in this example, the H100) is not part of the trusted computing base and therefore cannot access TD private memory. When the GPU raises a page fault on a non-resident page, the fault is first handled by the UVM driver (`nvidia-uvm.ko`), which determines the required migration. UVM then invokes the NVIDIA kernel driver (`nvidia.ko`) inside the TD to perform the necessary operations. The driver stages data through TD shared (bounce) buffers, performing a CPU-mediated memory copy between TD private memory and TD shared memory, followed by a DMA

transfer from shared memory to device memory (HBM). In this model, the driver acts as the trusted intermediary, introducing additional overhead compared to direct DMA in non-confidential environments.

In summary, in a TD guest:

- The CPU reads data from TME-encrypted private memory, which is transparently decrypted into clear text in the CPU cache.
- The data is then encrypted using the GPU's Security Protocol and Data Model (SPDM) session keys (for example, AES-GCM).
- The encrypted data is copied into TD's shared memory.
- The GPU Copy Engine (CE) fetches the data from shared memory and transfers it into GPU HBM, where it is decrypted by the GPU's security (SEC) processors.

CPU hotspot profiling shows [memcpy](#) and AES-GCM encryption as the dominant hotspots during data movement between the CPU and GPU.

Figure 4 below shows a step-by-step process to help understand this data movement between the CPU and GPU:

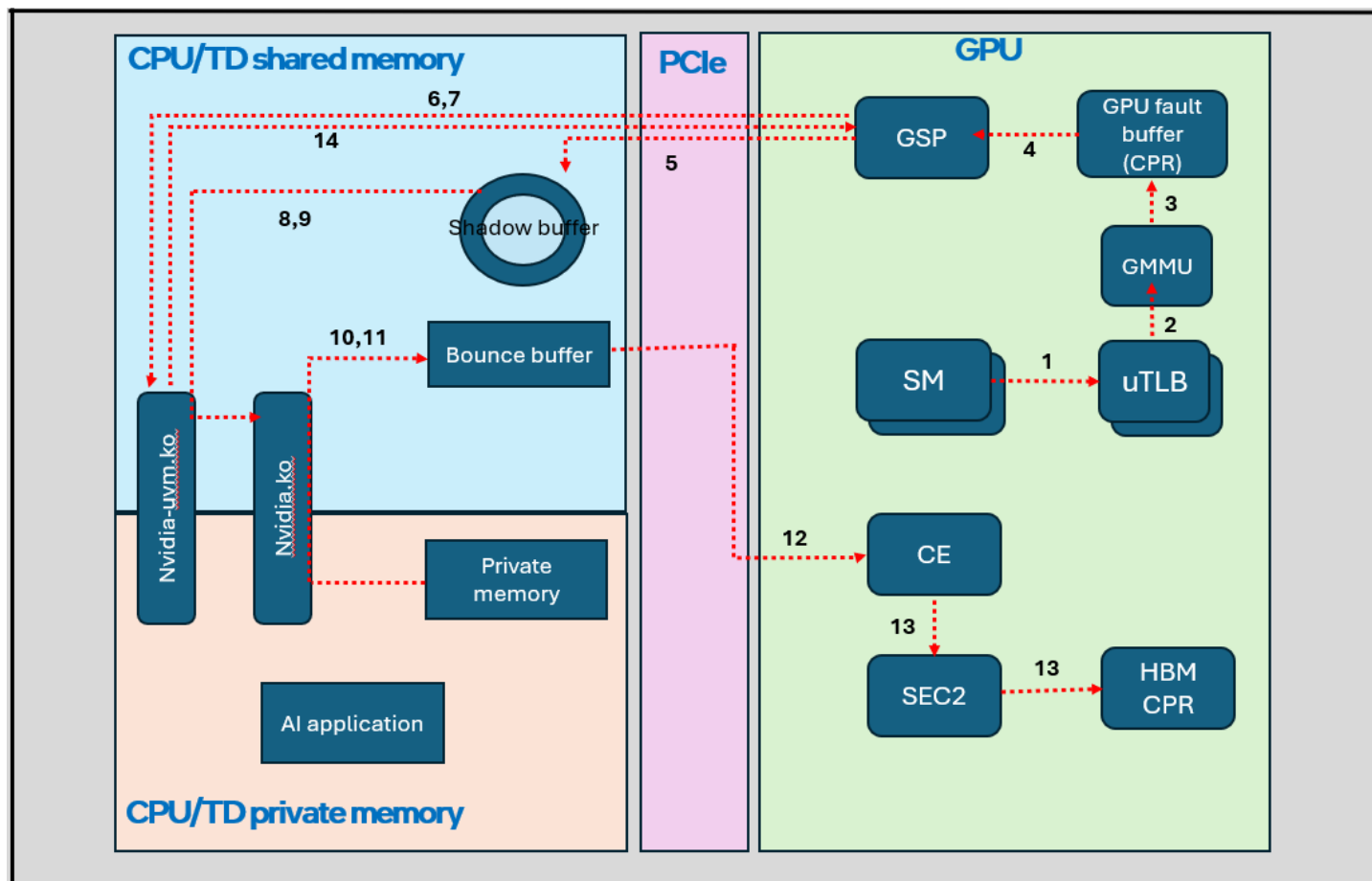


Figure 4: NVIDIA CC GPU-triggered page fault and data migration flow with UVM (derived from publicly available documentation)

Prior to runtime execution, the GPU confidential computing environment establishes a trusted communication channel between the CPU and GPU. Specifically, the SPDM requester within the NVIDIA kernel-mode driver initializes an SPDM session with the SPDM responder in the GPU System Processor (GSP). This session establishes a shared master secret, from which cryptographic keys are derived to protect GPU-CPU communication and data transfers across untrusted interfaces. With this secure context in place, subsequent GPU-CPU interactions—including fault reporting and data migration—occur over protected channels, providing confidentiality and integrity throughout the execution flow.

While both CPU- and GPU-initiated page faults are important, this section focuses on GPU-initiated page faults, which are the execution path described in the referenced work and examined here.

Phase 1: Hardware Detection & Blinded Reporting

1. SM Address Translation Attempt: A thread running on a Streaming Multiprocessor (SM) issues a memory access. The SM first looks up the address translation in its local uTLB (micro-TLB). If the translation exists, then execution proceeds. If the translation is missing, the request is forwarded to the GPU Memory Management Unit (GMMU) for address translation.
2. uTLB Miss → MMU Fault: Upon a uTLB miss, the GMMU performs a page table walk using the GPU page tables to resolve the virtual-to-physical address mapping. If the translation is found, the translation is inserted into the TLB hierarchy and execution resumes. If the translation cannot be resolved or the page is not resident, the GMMU generates a GPU page fault.
3. Fault Logging by GMMU: The GMMU logs the fault entry into the GPU Fault Buffer. In confidential computing mode, this buffer resides inside the GPU's protected region (CPR) and is not directly readable by the host.
4. GSP Secure Handling: The GPU System Processor (GSP) prepares the fault packet for secure transfer. In GPU confidential computing mode, replayable and non-replayable fault packets are encrypted by the GSP using dedicated fault keys (for example, [gsp_cpu_replayable_fault](#) or [gsp_cpu_non_replayable_fault](#)) before being exported to the host. The GSP is an embedded microcontroller inside modern NVIDIA GPUs (like the H100) responsible for device management tasks such as initialization, power and memory management, and coordination of confidential computing features. During initialization, the GPU and CPU establish authenticated communication channels (for example, via SPD) used to protect GPU–CPU control communications.
5. Encrypted Shadow Buffer Staging: The encrypted fault packet is copied into a shadow buffer in system memory for retrieval by host drivers.

Phase 2: The RPC Signaling and Host Intervention

6. Encrypted RPC Notifications: The GSP signals the host over the secure GPU–CPU communication channel that a fault packet has been queued by issuing an MMU_FAULT_QUEUED event through the GSP–host RPC mechanism.
7. UVM Driver handling: The UVM driver ([nvidia-uvm.ko](#)) running inside the TD receives the notification and schedules an interrupt service routine (ISR) to begin fault servicing. At this point, the UVM driver determines whether a memory migration or mapping update may be required to satisfy the GPU's request.
8. Decryption and Fault Metadata Retrieval: The UVM driver retrieves the encrypted fault packet from the registered shadow buffer in system memory. Using the corresponding fault key (for example, [gsp_cpu_replayable_fault](#) or [gsp_cpu_non_replayable_fault](#)), the driver verifies the authentication tag and decrypts the packet, after which it parses the fault metadata (such as the target virtual address) to continue fault handling.

Phase 3: Data Migration (The Hotspot)

9. Source Identification: The UVM driver determines that the requested page resides in the TD's private memory.
10. The Secure Migration: UVM initiates a migration. This is a performance hotspot:
 - The NVIDIA driver ([nvidia.ko](#)) is invoked by the UVM driver and performs a CPU-side memory copy of the page from private guest memory.
 - During the memory copy, the CPU encrypts the data (for example, using AES-GCM) with the SPD negotiated key.
11. Shared Bounce Buffer Staging: The encrypted data is written into a shared bounce buffer in system memory. Although this memory is not trusted, the payload remains protected because it is encrypted.

Phase 4: Resolution

12. DMA Transfer to GPU: The GPU's Copy Engine (CE) pulls the encrypted ciphertext over PCIe from the shared bounce buffer in system memory.
13. Decryption and VRAM Placement: The encrypted payload is decrypted on the GPU using the established session keys and placed into the GPU's protected memory region (for example, CPR) in HBM. *Refer to the disclaimer below regarding the exact hardware component responsible for decryption.*
14. Fault Replay: After migration completes, UVM sends an RPC to the GSP to replay the fault. The stalled GPU thread retries the memory access, the translation now succeeds, and execution resumes.

Disclaimer: Public documentation of NVIDIA GPU confidential computing describes cryptographic processing in multiple components of the GPU security architecture. The documentation indicates that the GPU Copy Engine (CE) maintains separate cryptographic keys for host-to-device and device-to-host data transfers, implying that encrypted memory transfers occur along the CE data path. The documentation also describes the GPU security processor (SEC2) performing cryptographic operations such as signature verification and decryption for protected payloads before storing them in the Confidential Protected Region (CPR). However, the documentation does not explicitly specify the microarchitectural data path responsible for encryption and decryption of runtime application data transfers, nor does it clearly isolate these operations to either CE or SEC2 alone.

Consequently, the data flow illustrated in Figure 4 should be interpreted as a conceptual representation of the control and data interactions among CE, SEC2, and protected GPU memory regions, derived from the behaviors described in the available documentation. Readers are referred to the original sources for detailed descriptions of the GPU confidential computing architecture.

5.0 Tier-2: GPU Runtime Execution Effects

In this section we will understand how Tier 1 architectural factors affect the GPU runtime execution like data and code characteristics.

When a kernel is launched, the runtime:

1. Configures a grid composed of thread blocks (groups of threads that execute together).
2. Sets up kernel arguments and ensures required data is available in device-accessible memory (via prior memory operations if needed)
3. Submits the kernel to the GPU for scheduling and execution on streaming multiprocessors (SM).

A CUDA graph extends this model by capturing a sequence of operations—such as kernels, memory copies, and synchronization steps—into a directed acyclic graph (DAG). A DAG is simply a dependency graph where each operation depends on previous operations but does not form cycles. Once captured, the entire graph can be launched as a single executable unit.

Graph execution reduces repeated launch overhead because the control setup is done once and then reused across multiple operations, rather than being repeated for every individual kernel launch.

5.1 Single-Kernel/Graph Queuing Behavior

We model the end-to-end lifetime of a kernel (or a graph execution instance) as:

$$T_{E2E} = T_{\text{prep}} + T_{\text{in}} + T_{\text{launch}} + T_{\text{exec}} + T_{\text{out}} + T_{\text{queue}} + T_{\text{sync}}$$

Each component corresponds to a distinct execution domain:

- T_{prep} — Host-side preparation time, including buffer allocation, data initialization, and any setup required before launching the kernel or graph.
- T_{in} — Data ingress (H2D), representing DMA transfer of input buffers from host memory to device memory.
- T_{launch} — Control-path latency required to enqueue and dispatch the kernel or graph to the GPU.
- T_{exec} — Active execution time of the kernel/graph on SMs including computation and memory accesses in HBM.
- T_{out} — Data egress (D2H), representing DMA transfer of output buffers from device memory to host memory.
- T_{queue} — Waiting time before execution begins due to prior work in the same stream or engine.
- T_{sync} — Host-side synchronization latency while waiting for device completion.

This decomposition separates service time (data movement + launch + execution) from waiting time (queueing + synchronization), enabling easier visualization of overhead. Figure 5 is a simple conceptual diagram that helps correlate these factors and does not represent measured timing proportions or quantitative relationships.

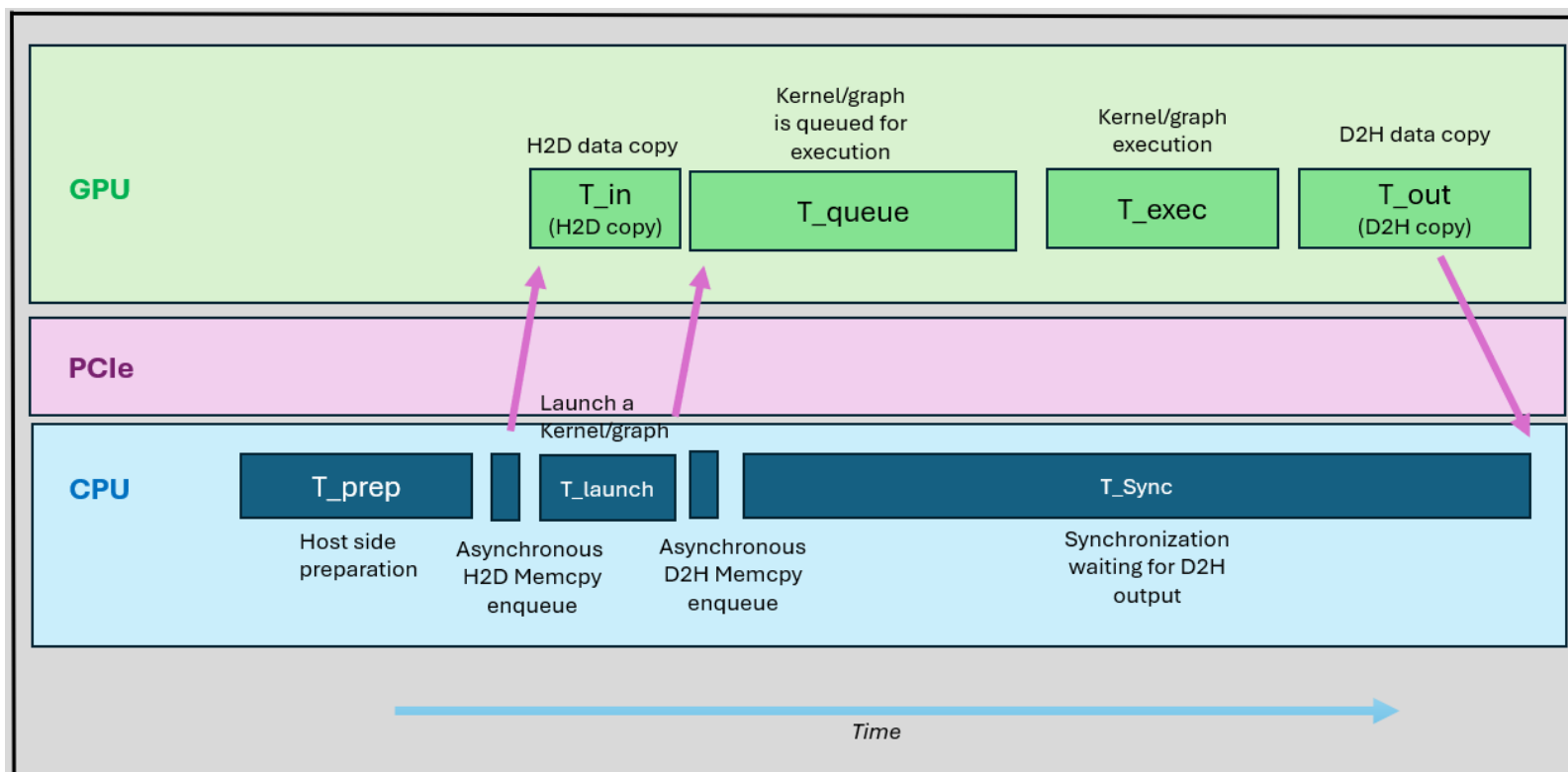


Figure 5: Conceptual diagram of kernel lifetime correlation

5.2 Multi-Kernel and Multi-Graph Queuing Behavior

In realistic AI workloads, kernels and graphs are rarely executed in isolation. Instead, multiple launches are enqueued into one or more CUDA streams. For kernel k , the effective start time depends on completion of prior work:

$$T_{\text{queue}}(k) = \max(0, C(k-1) - A(k))$$

Where:

- $A(k)$ is the time the kernel becomes ready for execution (after host enqueue),
- $C(k-1)$ is the completion time of the previous kernel in that stream.

The completion time is:

$$C(k) = A(k) + \Delta T_{\text{in}} + \Delta T_{\text{launch}} + \Delta T_{\text{exec}} + \Delta T_{\text{out}}$$

This formulation makes clear that any increase in data-path or control-path service time propagates forward, increasing queueing delay for subsequent kernels. Under high concurrency or long dependency chains (for example, graph execution with many nodes), queue amplification can dominate observed latency.

Figure 6 shows a sample cluster of kernel and graph execution observed under Nsight* Systems profiling with a 30B Llama model we will discuss further later. A similar clustering pattern was observed with other LLM model sizes studied in this paper as well. This visualization conceptually illustrates how asynchronous operations are queued up on the GPU for H2D or D2H data transfers. Kernels and CUDA graphs are subsequently launched and queued on the GPU, where they execute in the order defined by the stream and dependency structure. Once inference completes, the output is copied from the GPU back to the CPU, while the CPU blocks on a synchronization operation.

Disclaimer: Nsight profiling shows a prolonged `cuMemcpyAsync` duration; however, this may partially reflect profiling artifacts rather than actual sustained data transfer time.

Mapping to Tier-1 Mechanisms:

Tier-1 security factors primarily impact any data movement between the CPU and the GPU. Hence, they tend to increase the durations for:

$$\Delta T_{\text{in}}, \Delta T_{\text{launch}}, \Delta T_{\text{out}}$$

and occasionally contribute to ΔT_{prep} . These front-end delays can also propagate downstream as higher ΔT_{queue} and, depending on the workload and synchronization pattern, may be reflected in ΔT_{sync} . And most importantly, the arithmetic compute pipeline represented by ΔT_{exec} remains largely unchanged at the SM level as expected.

This means, with confidential computing, the tier-1 perturbations propagate as the following:

- Increased `memcpy` latency T_{in} and T_{out} due to encryption/decryption at CPU and GPU, bounce buffer related memory copy on the CPU, and other security operations mentioned before.
- Increased kernel launch duration T_{launch} due to the reasons mentioned above.
- Elevated stream synchronization time T_{sync} since the host must wait longer for the extended data movement and execution pipeline to complete.
- Potential reduction in effective GPU utilization, since the GPU is waiting for data to arrive and the kernel/graph to launch.
- Amplified queueing delay under concurrency T_{queue} , due to increase in T_{in} and T_{launch} .

These CUDA API duration increases reflect upstream changes in service-time components that alter the overall execution metrics like throughput and latencies.

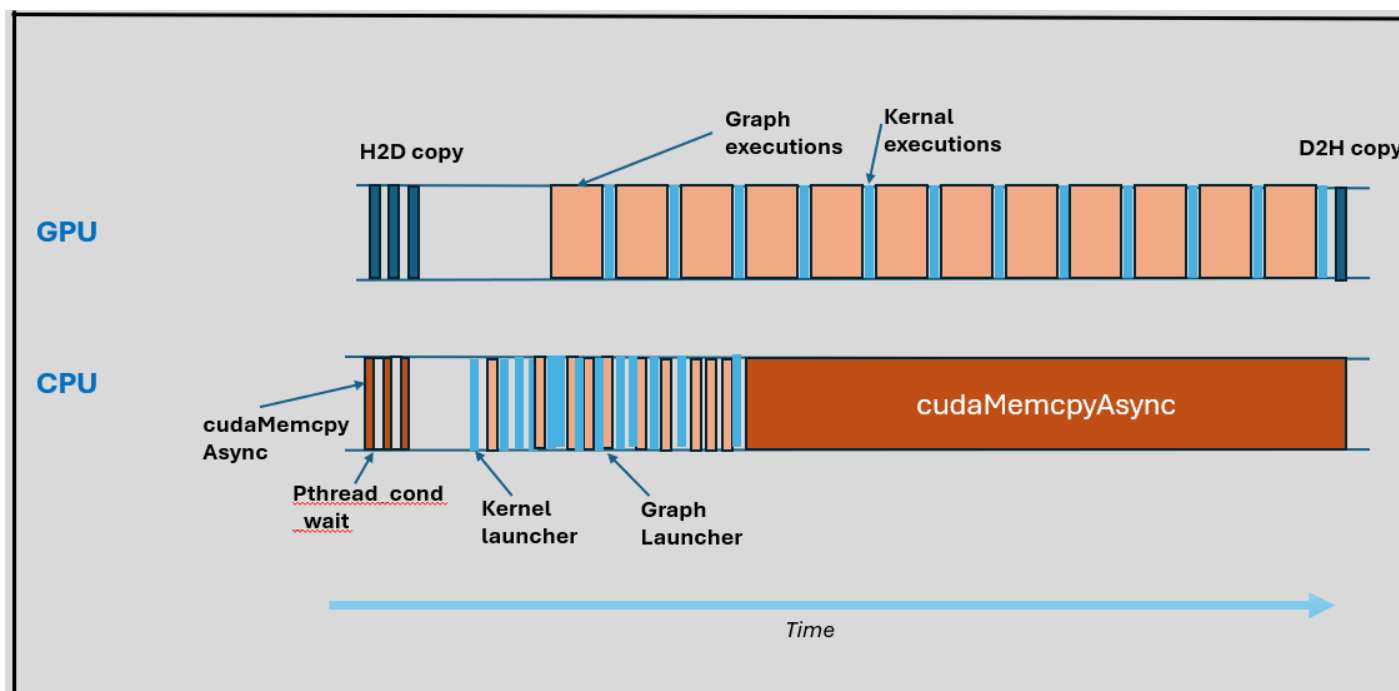


Figure 6. A sample kernel/graph cluster execution from 30B model.

6.0 Tier-3: System-level Performance Characterization

Building upon the runtime effects described in Tier-2, we now examine how confidential computing overhead manifests at the system level using end-to-end workload metrics and resource utilization indicators.

Throughput is measured in two ways:

- Requests per second (req/sec): The number of inference requests completed per unit of time.
- Output tokens per second (tok/sec): The total number of tokens generated per unit of time across all requests.

These two metrics capture different aspects of system behavior and are not necessarily interchangeable. Increasing the number of output tokens per request (that is, longer generations) will naturally reduce the number of requests per second because each request runs longer. However, the output token rate may remain constant or even increase, since more tokens are produced per request. Therefore, the req/sec metric reflects serving concurrency and request turnover, while tok/sec reflects sustained generation capacity.

Latency metrics include:

- Time to First Token (TTFT): The time from request submission to generation of the first output token, dominated by the prefill phase.
- Inter-Token Latency (ITL): The delay between consecutive output tokens during generation.
- Time per Output Token (TPOT): The average time required to generate each token during decode.

TTFT characterizes responsiveness, while ITL and TPOT reflect steady-state generation efficiency.

Inferencing contains two phases - prefill and decode. The prefill stage processes the entire input prompt in parallel, generating the initial key-value (KV) cache entries and producing the first output token. This stage is typically compute- and memory-intensive and largely determines Time to First Token (TTFT). The decode stage then generates output tokens iteratively, one token at a time. Each decode step reuses the KV cache and performs incremental attention computation. This phase determines sustained generation performance and directly impacts Inter-Token Latency (ITL) and Time per Output Token (TPOT). Together, these two stages define the end-to-end inference latency and throughput characteristics of the serving system.

In addition to workload metrics, we examine the following system resource indicators:

- CPU utilization, which may increase under confidential computing due to encryption, memory copies, and Intel TDX transitions.
- GPU utilization, representing the fraction of Streaming Multiprocessor (SM) capacity actively executing kernels.
- GPU memory utilization, including model weights and KV cache occupancy, which affects batching and concurrency limits.
- PCIe bandwidth utilization, which reflects data movement intensity between the CPU and GPU.

This paper assumes familiarity with system-level performance concepts such as resource saturation, compute-bound versus communication-bound behavior, and queueing effects. Accordingly, Tier-3 observations focus on cases where confidential computing meaningfully shifts system bottlenecks or alters expected scaling trends. Where workload behavior follows conventional GPU scaling principles, confidential computing overhead is interpreted as a bounded additive cost rather than as a structural limitation.

7.0 Performance Evaluation Studies

Building on the Tier-1 architectural analysis and Tier-2 runtime characterization, we now examine how these effects manifest at Tier-3 in observable workload behavior. Tier-1 defines the sources of confidential computing overhead, and Tier-2 explains how those mechanisms influence kernel launches, memory transfers, and synchronization paths. Tier-3 evaluates how these runtime perturbations translate into application-visible metrics such as throughput, latency, CPU and GPU utilization, memory occupancy, and PCIe bandwidth.

By applying this three-tier framework to representative workloads, we connect architectural mechanisms to system-level performance characteristics and determine when confidential computing overhead meaningfully influences workload behavior.

The platform used in the test environment of this paper is a dual-socket Intel® Xeon® 6 system (code named Granite Rapids) equipped with an NVIDIA H100 GPU (PCIe form factor) with 81,559 MiB of HBM memory, operating at a maximum SM frequency of 1755 MHz and an HBM frequency of 1593 MHz. The GPU is attached to one socket and is passed through directly to the guest VM. The VM and its memory are placed on the same socket as the GPU to ensure NUMA locality, while the OS and data disks reside on the other socket, allowing PCIe traffic from storage to be isolated from GPU traffic.

The test environment consists of an 8-vCPU guest with 32 GB of memory. Each vCPU is pinned to a dedicated physical core on the GPU-attached socket, and CPU frequency is fixed at 3.8 GHz (max turbo) to ensure consistent and repeatable performance. Guest memory is allocated from the same socket to avoid cross-socket access.

The software stack includes CUDA 12.8 and NVIDIA driver version 570.148.08. The host runs Ubuntu* 25.04 (kernel 6.14.0-1004-intel), while the guest runs Ubuntu 24.04.2 LTS (kernel 6.8.0-60-generic). Additional configuration details are provided at the end of this section.

Disclaimer: The reported results should not be interpreted as peak performance achievable on the platform. Performance is workload and system configuration dependent and may vary with different execution parameters. The goal of this study is to isolate, demonstrate, and analyze the performance overhead associated with confidential computing.

7.1 GPU-CPU Data Path Characterization with the NVBandwidth Tool

NVBandwidth is a CUDA microbenchmark tool designed to stress test the bandwidth between the CPU and the GPU, or between multiple GPU devices. It includes two types of tests: one that uses the GPU's Copy Engine (CE)—a dedicated DMA engine for data movement—and another that uses Streaming Multiprocessors (SMs), where CUDA kernels perform `memcpy` operations. Since CE-based transfers leverage dedicated DMA engines and are expected to achieve higher bandwidth with lower SM interference, we used the CE-based tests for our analysis. Use the `nvbandwidth -l` instruction to list all available tests.

For this study, we used tests 0 and 1, which are CE-based tests that copy data between CPU and the GPU asynchronously using `cuMemcpyAsync`, with a default buffer size of 512MB and default iterations of 3. We then compared the measured bandwidth for CPU-to-GPU transfers under a confidential guest environment compared to a non-confidential guest environment.

Disclaimer: Please note that this microbenchmark does not represent real-world AI inference behavior; instead, it isolates, stress-tests, and characterizes performance overhead related to CPU–GPU data movement in confidential computing environments.

7.1.1 Confidential Computing Performance Overheads

With confidential computing, the max bandwidth achieved between the CPU and the GPU is reduced to 13-15% of the non-confidential computing VM's performance, as shown in Figure 7. H2D (Host-to-Device) refers to data transfer from CPU memory to GPU memory, while D2H (Device-to-Host) refers to data transfer from GPU memory to CPU memory. While a non-confidential computing VM can transfer at ~55 GBps, a TD guest achieves ~ 8GBps bandwidth. This could potentially vary depending on the CPU and/or GPU frequencies and BIOS profiles.

Also, while a H2D/D2H test completes 3 iterations of the 512MB buffer copy within 95 seconds-175 seconds in a TD, they complete in just a few seconds in a non-confidential computing VM environment.

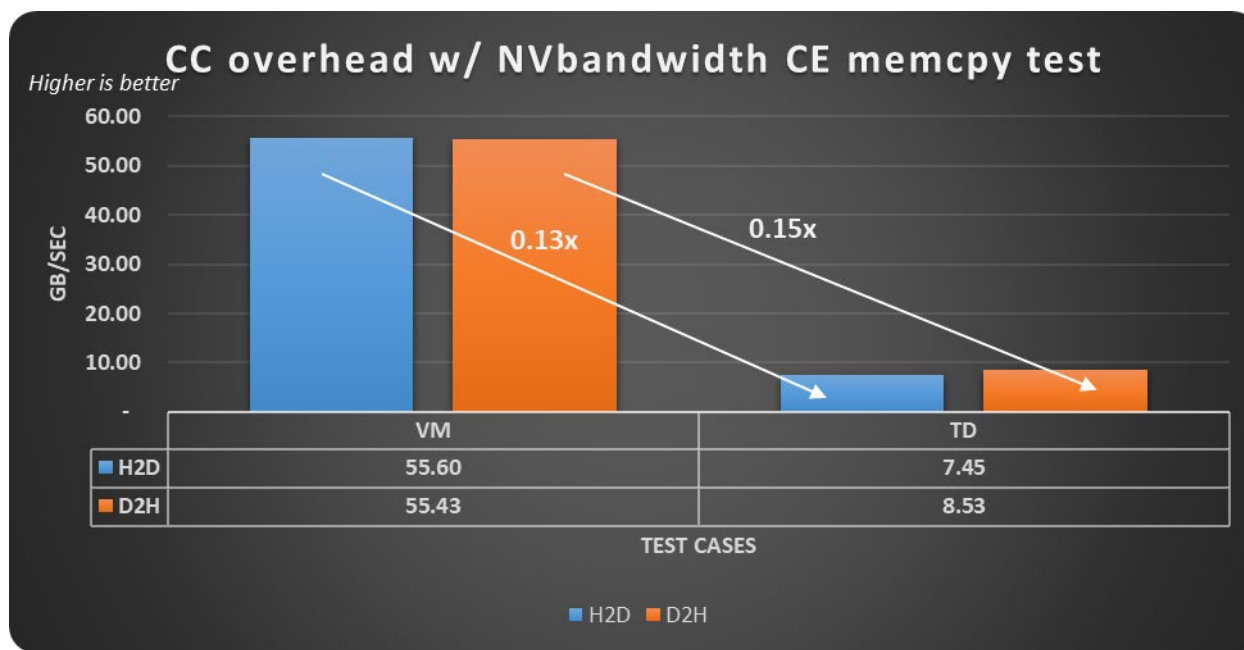


Figure 7: NVBandwidth memory bandwidth comparison for TD vs VM

7.1.2 Understanding the Workload

To understand confidential computing–related performance overheads for this workload, it is important to first understand how the workload operates. Execution of the CE-based tests in NVBandwidth proceeds through the following stages, based on Nsight profiling and code analysis:

1. Buffer allocation and initialization: After test parameters are determined, the `memcpyInitiator` function in `doMemcpyCore` allocates a host buffer using `cuMemHostAlloc` (pinned host memory in non-confidential guests). In confidential guests, UVM-managed memory is used instead. A device buffer is also allocated using `cuMemAlloc`. After allocation, the CPU launches `MemsetKernelDevice` in a CUDA stream to initialize the device buffers.
2. Stream synchronization for Initialization: While the GPU is allocating memory, the CPU is observed to be waiting for respective initialization/warmup operations that are queued up in the stream to complete via `cuStreamSynchronize`.
3. Asynchronous copy setup: Once initialization completes, `memcpyInitiator` calls `cuMemcpyAsync`, which sets up the DMA descriptors, programs the GPU copy engines, enqueues the data transfers into the stream, and returns immediately. While the GPU Copy Engines are preparing for the data copy, `spinkerneldevice` kernel runs on the SMs to enforce timing and ordering.
4. Data transfer: Copy Engines performs the actual transfer of data between CPU and GPU while `cuMemcpyAsync` queues up memory copy requests asynchronously on the device.
5. Validation: After the copy is completed, `memcpyPattern` is invoked for validation. For H2D copies, data is copied back to host memory and compared on the CPU. For D2H copies, a significant portion of validation time involves execution of `memcpyKernelDevice` on the GPU, operating on data fetched from CPU memory via UVM-managed page migration, as observed in Nsight profiling.
6. Memory deallocation: Finally, host and device memory are freed using `cuMemFreeHost` and `cuMemFree_v2`.

To comprehend the CC overhead, CC devtools mode was enabled with:

```
#nvidia_gpu_tools.py --set-cc-mode=devtools --reset-after-cc-mode-switch --gpu-bdf=<address>
```

And the Nsight Systems tool was run with:

```
#nsys profile -o /mnt/data1/trace/C2G_trace --trace=cuda,nvtx,osrt < function command>
```

Nvbandwidth was run with default iterations (3) for a TD, while it was run with 200 iterations for a non-confidential VM. Since the tool completed execution with the default iteration count in just a few seconds in the VM, we deliberately ran the VM longer, with iteration count of 200, to generate enough samples for analysis. For instance:

```
#!/root/nvbandwidth/nvbandwidth -t 0 -i 200
```

7.1.3 Tier-1 Architectural Performance Overhead Reasons

- CPU side confidential computing factors: CPU-level mechanisms such as TME have minimal impact on this workload and can be reasonably ignored. In addition, the TD's vCPUs and memory are collocated on the same NUMA node as the GPU device, eliminating UPI-related encryption overhead.
- Intel TDX transitions: As discussed earlier under Tier-1 VMM exit factors, Intel TDX increases the duration of each transition (TD-entry/TD-exit). For the NVBandwidth workload, total Intel TDX overhead increases primarily due to longer transition latency, while the total number of exits increases by approximately 10% due to page conversions from private to shared. This workload performs no disk I/O operations, so there are no additional bounce-buffer-related memory copies on the CPU.

- UVM and GPU related overheads: During bulk CE-based H2D and D2H transfers in confidential computing mode, approximately ~38% of one CPU core is consumed by cryptographic operations—encryption for H2D traffic and decryption for D2H traffic. In contrast, the additional [memcpy](#) contribution to overhead is less than 1%. Using OpenSSL 3, perf annotations show Intel® Advanced Vector Extensions 512 (Intel® AVX-512)–based vector AES instructions (for example, [vaesenc](#)) dominating the CPU hotspots.

7.1.4 Tier 2: GPU Runtime Execution Effects

Based on Nsight analysis, certain CUDA APIs executed by the CPU and functions run on the GPU took significantly longer time with confidential computing. Figure 8 shows a tabulation of these function durations based on execution order, as discussed in the previous section. The table compares these functions for a TD vs a VM for both Host to Device CE (H2D) copy and Device CE to Host (D2H) copy.

Function durations show increase between 100x to 124Kx due to confidential computing. And data copy operations for D2H and H2D show an increase of ~7x.

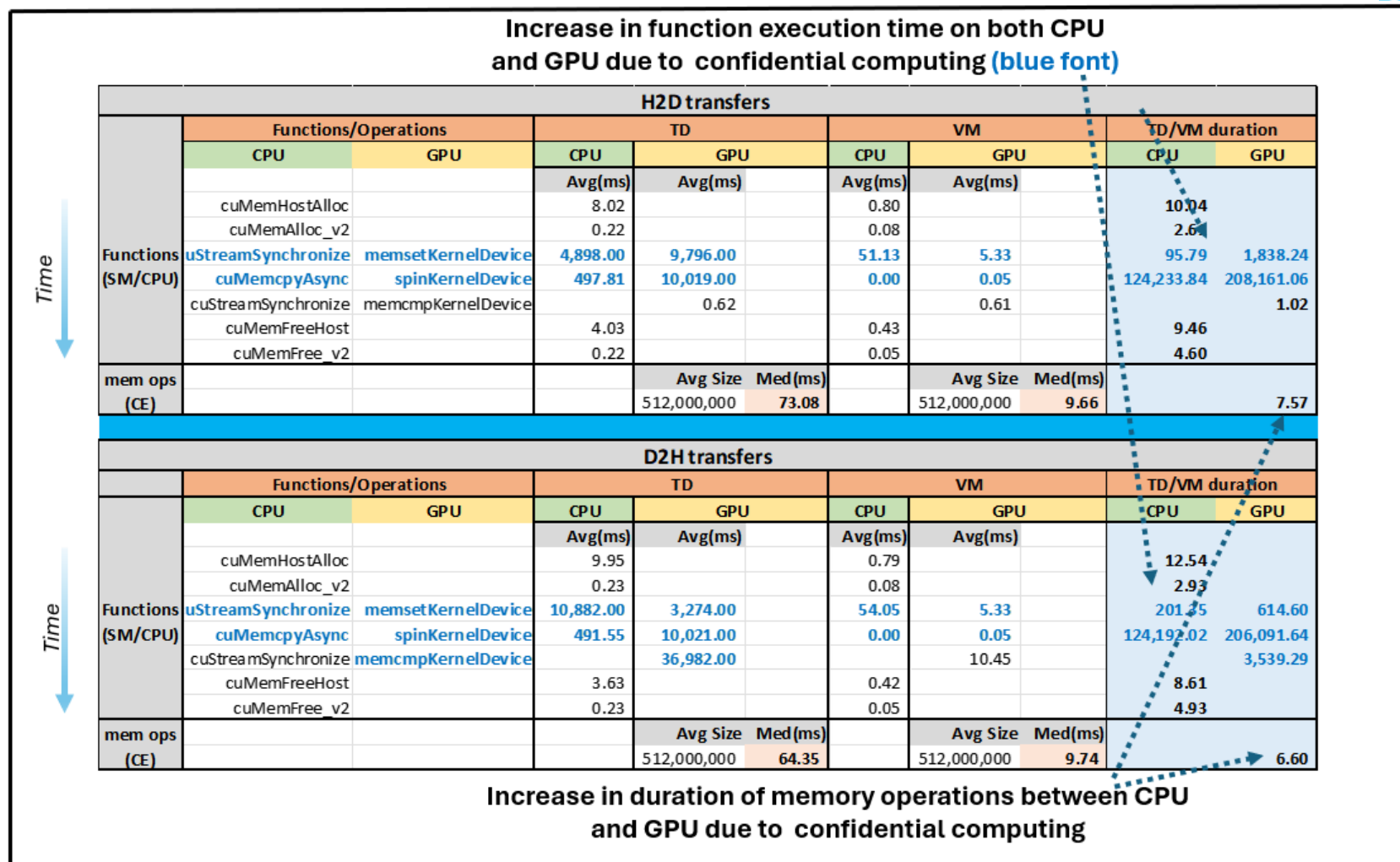


Figure 8: CUDA API and GPU execution metrics for TD vs VM

Based on Nsight Systems analysis, the functions found in the critical path are the CUDA APIs `cuStreamSynchronize` and `CuMemcpyAsync`, and GPU functions `memsetKernelDevice` and `spinKernelDevice`. We explain why each of these critical functions take longer with confidential computing below:

- **`memsetKernelDevice`:** The GPU executes this function to initialize its device buffer pages. When the GPU accesses a virtual address that is not yet backed by a physical HBM page, a page fault is generated and sent to the CPU. On the CPU side, UVM allocates a new HBM page corresponding to the IOVA and initializes it to zero for security purposes. As part of the initialization, the zero filled DRAM page is copied into the GPU's HBM page, which leads to several H2D transfers. As per Nsight, the total size of these transfers is about 512 MB, the size of the workload buffer. Due to

confidential computing, these H2D transfers increase the overall duration of the `MemsetKernelDevice` function because each transfer must undergo CPU-side and device-side security and integrity operations.

- **`cuMemcpyAsync`:** This is a CUDA API executed by the CPU to enqueue memory copy operations on the GPU. The actual data movement is performed by the GPU's Copy Engines (CE) to DMA transfer data from the CPU to the GPU or vice versa depending on application parameters. In confidential computing mode, the device buffer pages owned by the GPU may need to be accessed or validated by the CPU for various purposes, such as DMA setup or confidential computing checks. This causes page faults that are triggered by the CPU/UVM and are seen as D2H page faults in Nsights for 512 MB buffer size. This D2H related overhead is observed in the first `CuMemcpyAsync` operation (out of the 20) queued up on the GPU, which in turn slows down the overall DMA memory transfer.

Asynchronous copies, as the name indicates, do not wait for the DMA transfers to be completed before the function returns. In a non-confidential guest, the DMA transfers are observed to start after all the 20 asynchronous copies are finished queuing up. In a confidential guest, additional page fault handling and security checks introduce delays during the initial copy setup. As a result, DMA transfers may begin earlier relative to subsequent enqueues, because the CPU-side overhead of the first copy increases submission latency. This partially overlaps DMA execution with host-side queuing and can slightly reduce the overall measured workload duration compared to a fully serialized enqueue pattern.

- **`memcpyKernelDevice`:** The GPU executes this validation function to verify that the data transferred from the device to the host is correct. This validation phase is observed primarily when the workload performs D2H DMA transfers. As part of validation, the GPU may access data residing in host memory; if the data is not resident in GPU memory, this triggers GPU page faults and subsequent H2D migrations to bring the data back into GPU memory for comparison. Nsight profiling confirms that these H2D transfers incur higher overhead in confidential computing mode, as they must undergo additional encryption, integrity checks, and page-handling operations. This added overhead increases the total execution time of the validation function, contributing to the overall performance impact observed under Intel TDX.
- **`cuStreamSynchronize` and `spinKernelDevice` functions:** When the GPU executes the `memsetKernelDevice` function mentioned earlier, the CPU calls `cuStreamSynchronize` waiting for the `memset` kernel and the associated H2D transfers to complete. Similarly, when the `cumemcpyAsync` API is invoked on the CPU, the GPU executes `spinKernelDevice` to serialize asynchronous copies for the Copy Engines and to enforce ordering constraints within the stream. In confidential computing mode, the increased duration of `memsetKernelDevice` and `cuMemcpyAsync`—due to additional H2D and D2H-related page handling, encryption, and integrity operations—propagates into longer synchronization intervals. As a result, both `cuStreamSynchronize` and `spinKernelDevice` exhibit increased execution time.
- **Memory operations:** Lastly the H2D or D2H DMA copy operations performed by the Copy Engines take ~7x longer in TDs, which can be attributed to the factors described in the Tier-1 architectural performance overhead reasons section above.

In summary, NVBandwidth profiling indicates that confidential computing consistently increases the duration of data movement primitives such as `cuMemcpyAsync`, `memset`, and `memcpy`. These overheads stem from both explicit DMA transfers and page fault-driven migrations, which introduce additional staging and CPU-side processing in TD environments, ultimately increasing execution latency and reducing effective bandwidth.

7.1.5 Tier-3 System-level Performance Characteristics

NVBandwidth executes as a process pinned to one CPU core, which reaches nearly 100% utilization during execution. Under confidential computing, data movement operations observed on the GPU—such as H2D and D2H transfers triggered by DMA or page faults—are extended due to additional staging and CPU-side processing, including encryption, decryption, and `memcpy`. These operations increase the per-transfer latency, which manifests as longer `cudaMemcpyAsync`, `memset`, and `memcpy` durations on the GPU timeline.

When two NVBandwidth instances are run concurrently, aggregate CVM–GPU throughput scales from approximately 7.5 GB/s to 15 GB/s. This scaling indicates that the observed delays are not limited by PCIe bandwidth or GPU capacity, but by CPU-side processing. Since each instance can utilize a separate CPU core, parallelizing the cryptographic and data movement operations reduces contention and improves overall throughput. Refer to section 8 for comprehensive performance optimization takeaways.

7.2 Model Capacity Scaling Analysis with vLLM

7.2.1 Understanding the vLLM Serving Benchmark

vLLM is an open-source, high-performance GPU inference engine built on PagedAttention, which manages the KV cache using a paged, non-contiguous layout within GPU memory (HBM). Here, “paging” refers to logical memory management within the GPU, not OS-level paging or CPU–GPU data migration, enabling improved memory efficiency, throughput, and latency.

The vLLM serving benchmark, built around the vLLM engine, is a performance evaluation framework used to measure LLM inference efficiency under realistic, high-concurrency workloads. This benchmark focuses on end-to-end serving behavior rather than just kernel-level performance. It reports metrics such as:

- Throughput: output token throughput (tokens/sec) and request throughput (requests/sec)
- Latency: Time to First Token (TTFT), Time per Output Token (TPOT), and Inter-Token Latency (ITL)

In our experiments, the vLLM serving benchmark uses prompts derived from the ShareGPT_V3 dataset, obtained from the publicly available ShareGPT_Vicuna_unfiltered dataset hosted on Hugging Face.

In this section, we analyze workload characteristics and confidential computing overheads across increasing model sizes of the LLaMA and InternLM2 families listed in Table 1. LLaMA is a family of open-weight large language models developed by Meta* and is widely used as a foundational architecture for research, fine-tuning, and inference benchmarking. InternLM2 is an open-source model family developed by the Shanghai AI Laboratory, emphasizing reasoning and long-context capabilities. It is often released in LLaMA-compatible formats, enabling reuse of existing inference and serving infrastructure.

The NVIDIA H100 PCIe- formfactor GPU deployed in the test bed has nearly 80 GB of HBM memory size, and the model sizes selected below scale from 1B (that is, 2 GB of GPUs memory) up to HBM memory capacity saturation.

Model	Modality	Parameters	Data Type / Precision	Approx. Model Memory	Reference Link
Llama-3.2-1B	Text	~1B	FP16 / BF16	~2 GB	https://huggingface.co/meta-llama/Llama-3.2-1B
Llama-3.2-3B	Text	~3B	FP16 / BF16	~6 GB	https://huggingface.co/meta-llama/Llama-3.2-3B
Llama-3.2-11B-Vision-Instruct	Vision + Text	~11B	FP16 / BF16	~22–24 GB	https://huggingface.co/meta-llama/Llama-3.2-11B-Vision-Instruct
internlm2_5-20b-llamafied	Text	~20B	FP16 / BF16	~40 GB	https://huggingface.co/internlm/internlm2_5-20b
llama-30b-instruct	Text	~30B	FP16 / BF16	~60 GB	https://huggingface.co/meta-llama
Llama-3.3-70B-Instruct (W8A8) (quantized model)	Text	~70B	INT8 (W8A8)	~70–80 GB	https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct

Table 1: AI models used in increasing sizes

7.2.2 Confidential Computing Performance Overheads

Based on our observations, as the model size scales, the relative confidential computing (CC) overhead decreases. The 1B parameter model experiences approximately 10% performance drop, while larger models such as 20B, 30B, and 70B (quantized) show a reduced overhead of about 3%, as shown in Figure 9. Similarly, CC increases output token latency metrics, including Time to First Token (TTFT), Inter-Token Latency (ITL), and Time per Output Token (TPOT). For the 1B model, these latencies increase by approximately 10%, whereas for larger models (20B, 30B, and 70B), the increase is reduced to around 2% as shown in Figure 10.

As model size increases, vLLM throughput decreases and latency increases due to higher per-token compute cost. This is driven by a larger number of transformer layers, increased memory bandwidth pressure, and larger KV-cache footprints, which can limit effective batch size and reduce scheduling efficiency. As token round-trip latency (measured by TTFT and TPOT) increases, the relative overhead introduced by confidential computing decreases. This is because fixed CPU- and GPU-side overheads—such as encryption, additional memory handling, and control-path costs—are diminished over longer compute-dominated execution intervals. Provided other parameters remain constant, the proportional performance impact of confidential computing therefore reduces as model size grows.

Further specifics are discussed in the sections below.

As for the workload parameters, vllm serving was run with a maximum sequence number `--max_num_seqs` of 16 and max size of input and output token count `--max_model_len` of 2048 – 4096 depending on the what the model could support. The benchmark was run with `--num-prompts` 500 for longer run and stable results.

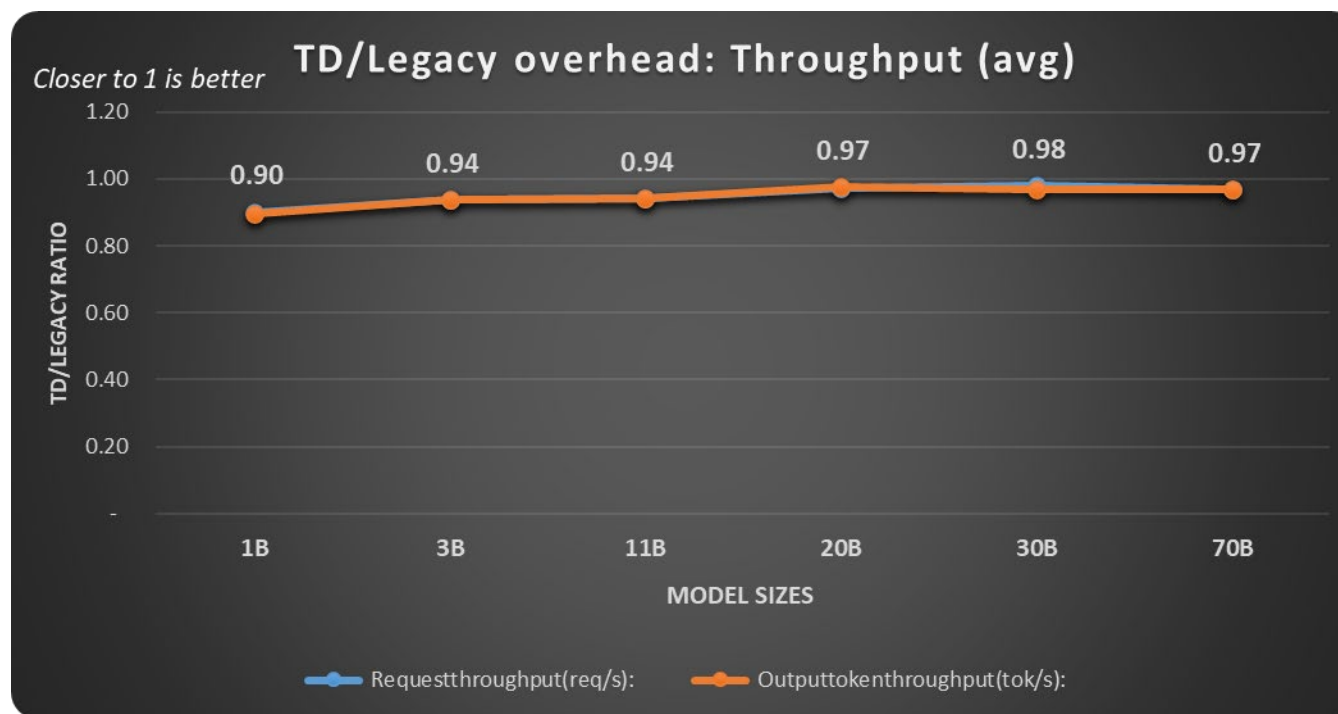


Figure 9: Throughput overhead due to confidential computing

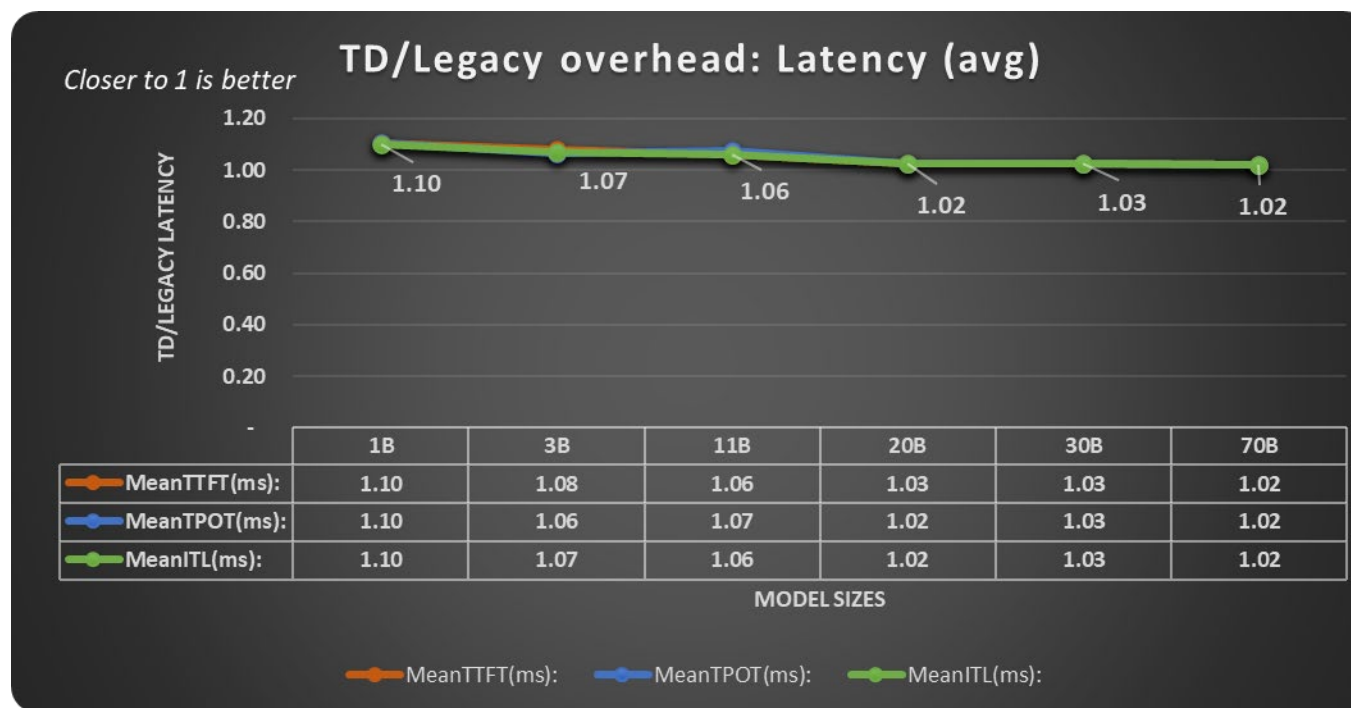


Figure 10: Latency overhead due to confidential computing

The vLLM benchmark consists of two phases: vLLM serving and vLLM inference. The serving phase initializes the model and runtime environment, while the inference phase processes requests and generates output tokens, forming the performance-critical portion of the workload.

7.2.2.1 vLLM Serving

The vLLM serving benchmark execution consists of two phases: serving and inferencing. The serving phase occurs before inference begins and is not part of the critical performance path measured in this study. However, it is useful to briefly describe its role.

vLLM serving involves loading the LLM weights into GPU memory once and exposing the model as a high-throughput, low-latency service. During this phase, the engine initializes model state, allocates GPU memory, and prepares runtime structures. Once initialized, the serving engine handles multiple inference requests by performing dynamic scheduling, batching, and efficient KV-cache memory management.

During the serving phase, the vLLM engine initializes using the configuration and runtime parameters specified by the user. The scheduler and memory managers are set up to handle batching and KV-cache allocation. Next, the model is loaded from disk. The models described earlier are stored as multiple checkpoint shard files. Each checkpoint shard is read from disk into CPU memory and then transferred to GPU memory before the next shard is read. This process continues until all shards are loaded and the full model is resident in GPU memory.

Additional overhead is observed during the serving phase due to confidential computing, primarily due to following reasons as discussed under Tier-1 section:

- Shared ↔ Private memory copies for virtual devices: Model checkpoint shards are read from disk through virtualized storage devices. In a TD, data received via virtual devices must first reside in shared memory and then be copied into the guest's private memory. This shared-to-private memory copy introduces additional CPU overhead.
- UVM-related overhead during CPU→GPU transfers: When model weights are transferred from CPU memory to GPU memory, UVM and NVIDIA driver paths introduce additional overhead due to encryption and staging through shared bounce buffers. Data must be encrypted on the CPU before being DMA-transferred to the GPU, increasing CPU utilization during model loading.
- Increased KVM / Intel TDX exits: Confidential computing introduces additional and/or longer exits due to page conversions, shared memory handling, and Intel TDX transitions.

Following model loading, vLLM performs KV-cache allocation and CUDA Graph capture. The server reserves a portion of GPU memory for the KV cache and then executes a dummy inference pass to trigger kernel compilation and CUDA Graph capture.

Using the 30B model as an example: when loaded in bfloat16 format, the model weights occupy approximately 60 GB of GPU memory. The server then allocates approximately 8.75 GB for the KV cache and an additional ~1 GB for CUDA Graph objects. Together, these allocations nearly saturate the 80 GB HBM capacity of an NVIDIA H100 GPU. This aggressive memory utilization leaves minimal free HBM headroom, making memory management efficiency critical during subsequent inference.

7.2.2.2 vLLM Inference

After the serving phase completes, the system enters the inference phase, which lies on the critical performance path. During inference, the vLLM engine processes incoming user requests by executing the model in two main stages: *prefill* and *decode* as briefly mentioned under the Tier-3: System-level Performance Characterization section.

Disclaimer: Due to limited visibility in Nsight profiling, we are unable to clearly separate the prefill and decode phases at the GPU kernel level. While the conceptual distinction between prefill and decode is important, the profiling traces do not always allow unambiguous classification of individual kernels or memory transfers into these two categories. Therefore, in this study, we treat inference as a single unified phase and analyze its performance characteristics in aggregate. All reported metrics (throughput, TTFT, ITL, and TPOT) reflect end-to-end inference behavior without explicitly decomposing GPU activity into prefill and decode components.

Now let's analyze the performance characteristics of the performance-critical inference phase of the workload across Tier-1, Tier-2, and Tier-3.

7.2.3 Tier-1 - Architectural Performance Overhead Sources

The following factors were discussed in detail under the Tier-1 section.

- CPU side confidential computing factors: CPU-level mechanisms such as Intel TME have minimal impact on this workload and can be reasonably ignored. In addition, the TD's vCPUs and memory are collocated on the same NUMA node as the GPU device, eliminating UPI-related encryption overhead.
- Intel TDX transitions: The nature of exits observed in the confidential guest during inference is similar to those discussed earlier for the NVB bandwidth workload. Overall exit rates for a TD are only slightly higher than those of a non-confidential VM, as shown in Figure 11. The primary increase in exit count is attributed to APIC ICR MSR write operations used for IPI delivery, often triggered by page conversions and TLB shutdowns.

In addition to the modest increase in exit rate, each Intel TDX transition incurs higher latency compared to a traditional VMX transition. As a result, even with comparable exit frequencies, the longer transition duration in confidential guests leads to a higher overall performance impact from KVM exits during inference.

KVM stats	TD	VM
kvm_exit	7,653	7,642
kvm_exit(EXTERNAL_INTERRUPT)	2,429	1,271
kvm_exit(HLT)	1,014	1,136
kvm_exit(PREEMPTION_TIMER)		1,158
kvm_exit(MSR_WRITE)	4,210	4,065
MSR 6e0 (IA32_TSC_DEADLINE)	3,395	3,298
MSR 830 (IA32_X2APIC_ICR)	336	8

Figure 11. TD vs VM KVM exit comparison for 30B vLLM inferencing

- UVM and GPU related overheads: With confidential computing, CPU profiling shows overheads in two paths:

7.2.3.1 The Secure Command Path

In Confidential Computing mode, the GPU is considered untrusted. When the `cudaLaunchKernel` or `cudaGraphLaunch` functions are invoked, the device driver running on the CPU cannot transmit kernel launch commands in plaintext over the PCIe bus, as they could potentially be observed or modified. Therefore, the CUDA driver encrypts the kernel launch commands (pushbuffer contents) and associated kernel arguments before they leave the CPU. CPU hotspot profiling shows the `vaes_gcm_cipherupdate` function is used to perform encryption and integrity protection on the CPU side before sending the kernel/graph launch commands to the GPU, which in turn uses hardware-accelerated Intel® Vector AES (Intel® VAES) instructions for cryptographic processing.

7.2.3.2 Secure Data Path via Bounce Buffers

In confidential guests, the CPU and GPU cannot directly share plaintext memory. As a result, UVM triggers page migration between system memory (DRAM) and GPU memory (HBM), introducing additional overheads associated with confidential computing, as discussed in the Tier-1 section.

Only a small amount of input and output data (on the order of bytes) is transferred between host and the device during inference. Therefore, overhead due to UVM/CC related encryption/decryption and memory copies is very small, on the order of 1-2% for the 30B model. This leads to a corresponding performance drop due to confidential computing.

7.2.4 Tier-2: GPU Run time Execution Effects

To comprehend confidential computing overhead, we should look at the following three aspects. While overheads due to UVM enc/dec/Memcpy and KVM exits can slow down operations on the CPU side, security operations performed by the GPU processors like SEC and CE can slow down GPU-side operations, as discussed below.

7.2.4.1 H2D and D2H Memory Transfer Time

This is observed to increase 2x due to security and integrity operations mentioned before, as measured by the Nsights profiling shown in Figure 12.

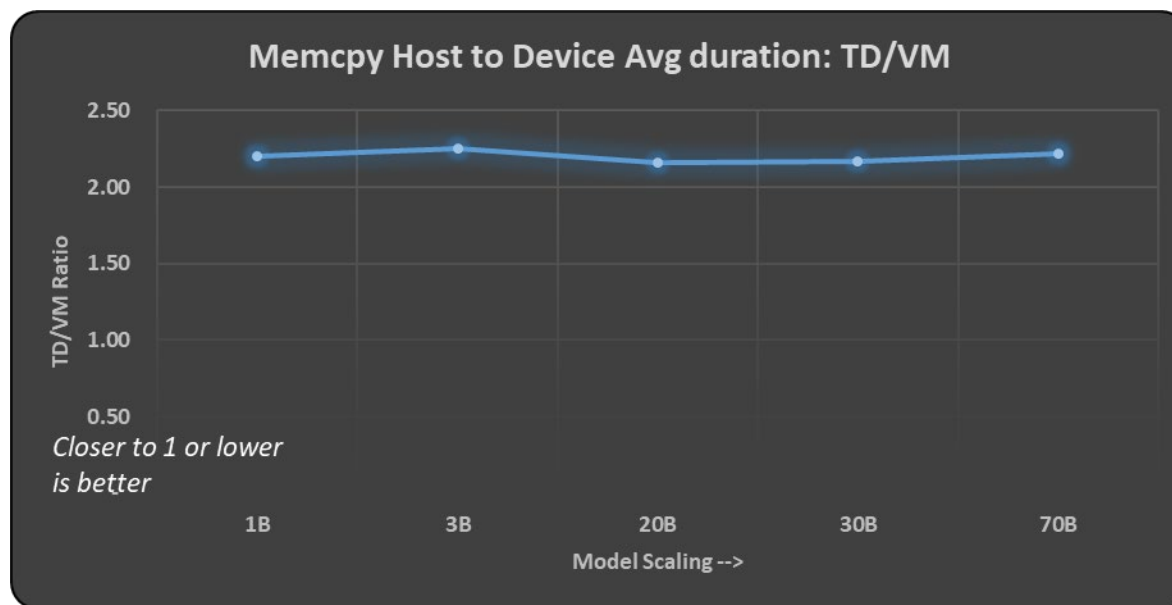


Figure 12: TD/VM comparison of memory operation duration at the GPU

7.2.4.2 Kernel Launch Time

Nsight Systems profiling shows that CUDA APIs like the `cudaLaunchKernel` and `cudaGraphLaunch` functions invoked by the CPU take a longer time with confidential computing. As mentioned before, these overheads can be attributed to encryption via `vaes_gcm_cipherupdate` like functions that help encrypt kernel launch commands and arguments via the push buffers.

Figure 13 shows CUDA APIs related to kernel and graph launches, which are the API's taking the longest durations during inference, extending further up to 5x due to confidential computing. `cudaMemcpyAsync` is an asynchronous API that doesn't need to wait until completion of execution and hence isn't necessarily relevant to this comparison. We should consider the possibility of a small margin of error introduced by profiling itself.

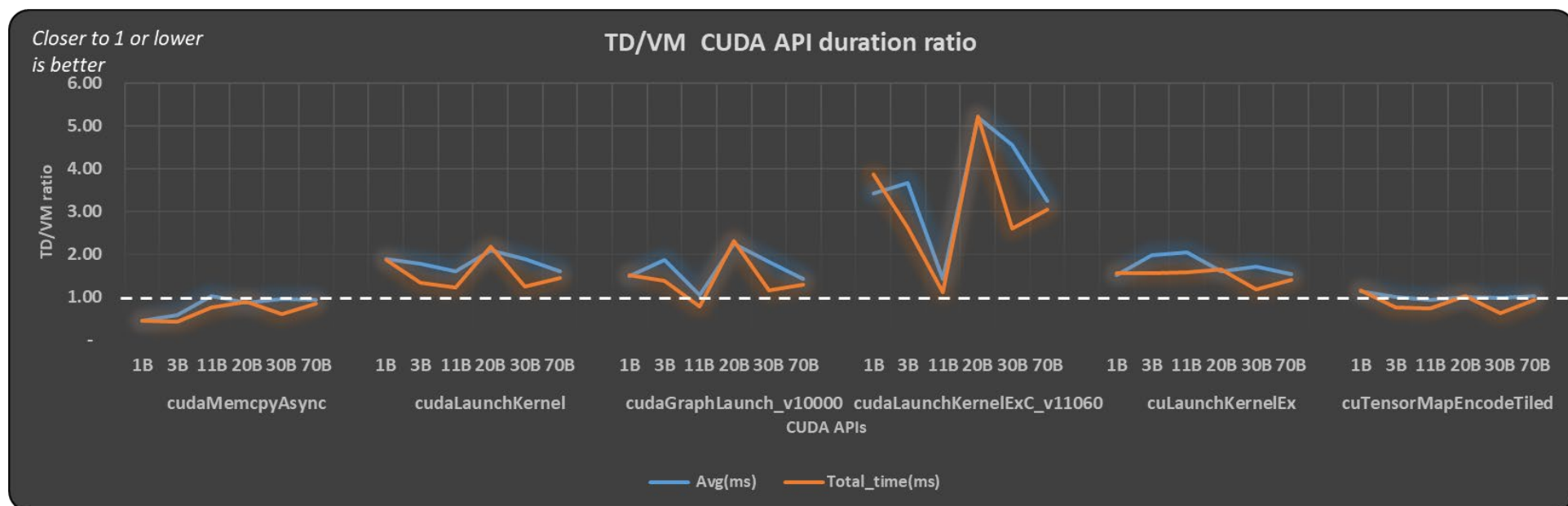


Figure 13: TD/VM CUDA API duration ratio

7.2.4.3 Kernel execution time

Confidential computing does not impact kernel or CUDA graph execution durations. As shown in Figure 14, the execution time of GPU kernels remains unchanged under confidential computing. This is expected because these kernels operate on GPU-resident memory—L1, L2, and HBM—which lies within the trusted computing base (TCB) of the confidential guest and therefore does not incur additional PCIe-level encryption overhead. Since execution within the GPU's compute and memory hierarchy does not incur additional cryptographic overhead, kernel runtime remains consistent between confidential and non-confidential modes.

This observation also explains why confidential computing overhead decreases as model size increases. Larger models spend proportionally more time executing kernels and graphs on the GPU (that is, in compute-bound execution), while the relative contribution of CPU/GPU-side security related overhead—such as encrypted kernel launches and CPU–GPU data transfers—becomes smaller. As a result, the overall performance impact of confidential computing is reduced for larger, more compute-intensive workloads.

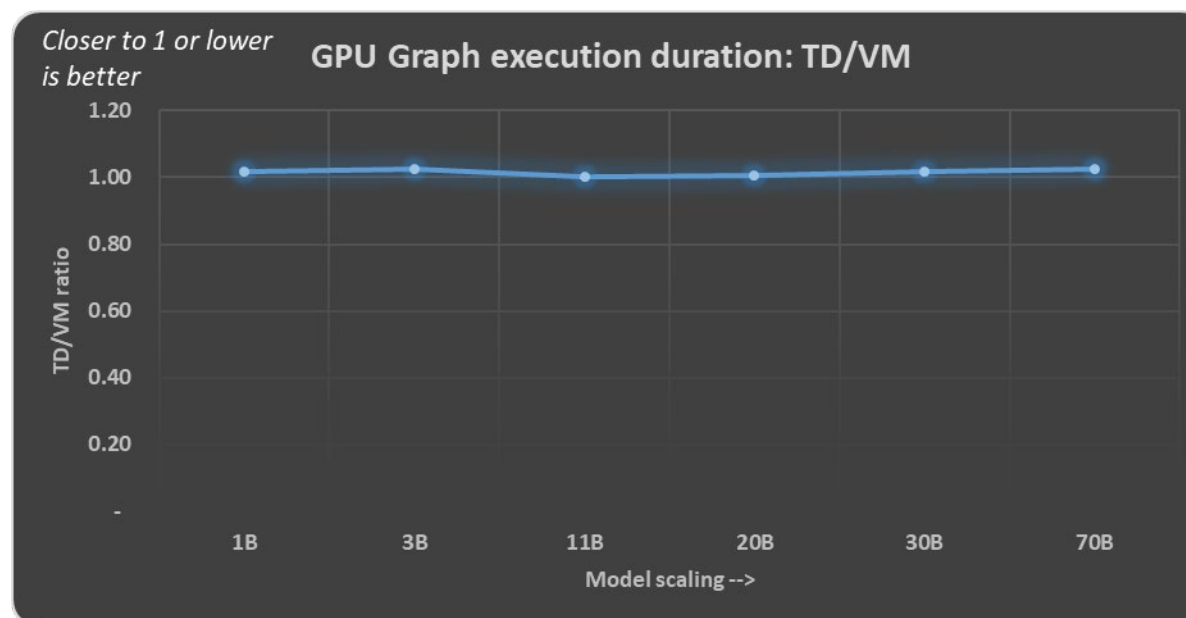


Figure 14: TD/VM Graph execution duration ratio

Hence increase in ITL, TPOT and TTFT are expected to increase with confidential computing.

7.2.5 Tier-3 System level performance characteristics

- CPU profiling shows 1 vCPU running vLLM serving at 100% saturation. This process is responsible for launching kernels and graphs on the GPU for execution and transferring input data from Host to Device and output data from Device to Host, interspersed with pthread conditional wait functions. Hence any security operations happening on that vCPU can impact workload performance.
- GPU profiling shows a very small impact on SM utilization as well, where SM is occasionally seen 1-2% less utilized, especially in smaller models as shown in Figure 15. If the graphs/kernel execution is waiting on arrival of input data from the CPU, any delay due to UVM CC overheads, can lead to SM being idle waiting for the functions and data to execute. Figure 15 also shows a similar drop in GPU power consumption that we observed with confidential computing.

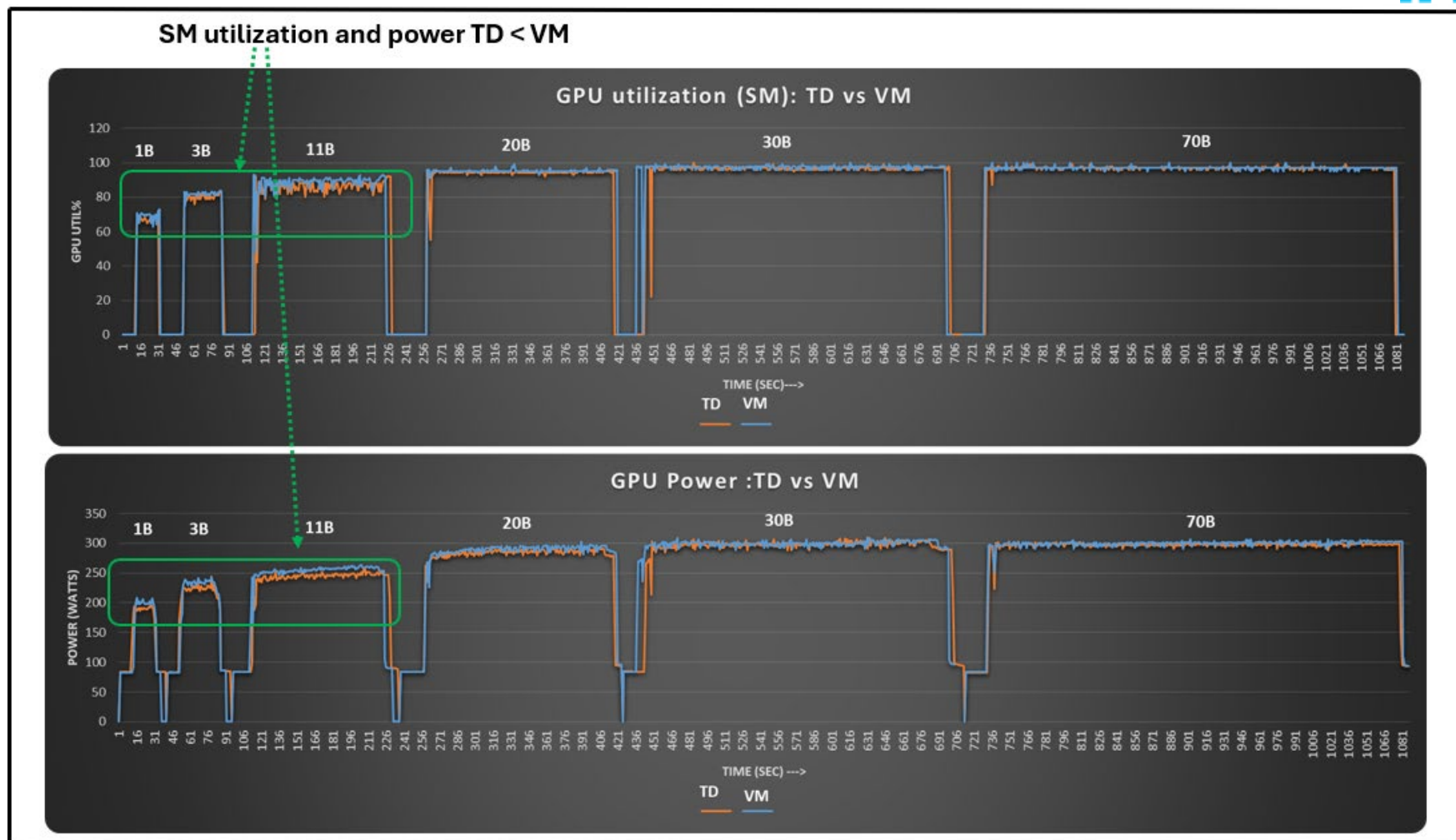


Figure 15: GPU utilization and power comparison for TD vs VM

Disclaimer: In our experiments, we found that the GPU tends to hit power throttling with large models. We observed this with 20B, 30B, and 70B models, quantized with both confidential and non-confidential guests. This issue was reproduced with multiple H100 GPUs of different models, using the latest driver/ucode versions. Refer to system setup section at the end for GPU configuration.

In summary, as model size increases, inference becomes more compute-dominated due to higher per-token processing cost and longer GPU execution time. While confidential computing overhead remains present, its relative impact decreases because fixed control-path and data-movement costs become smaller compared to sustained GPU computation. Consequently, larger models exhibit lower proportional performance degradation than smaller models.

7.3 Concurrency and Sequence-length Scaling Analysis with vLLM

Beyond model sizes, several other factors can affect confidential computing performance overhead. To demonstrate this, we scaled three workload parameters, max concurrency, input length, and output length, using a Llama-3.2-1B model and a “random” dataset, as shown below in Figure 16.

`--max-concurrency` defines the maximum number of concurrent requests that will be allowed to execute at a time.

`--input-len` defines number of tokens per input dataset.

`--output-len` defines number of tokens per output dataset.

Our study shows that as these factors increase, performance overhead due to confidential computing decreases.

Disclaimer: Note that the GPU is observed to hit power throttling at max concurrency of 512 and higher. This leads to fluctuations in latency metrics like ITL, TPOT, etc., though the throughput and duration metrics remain stable run-to-run.

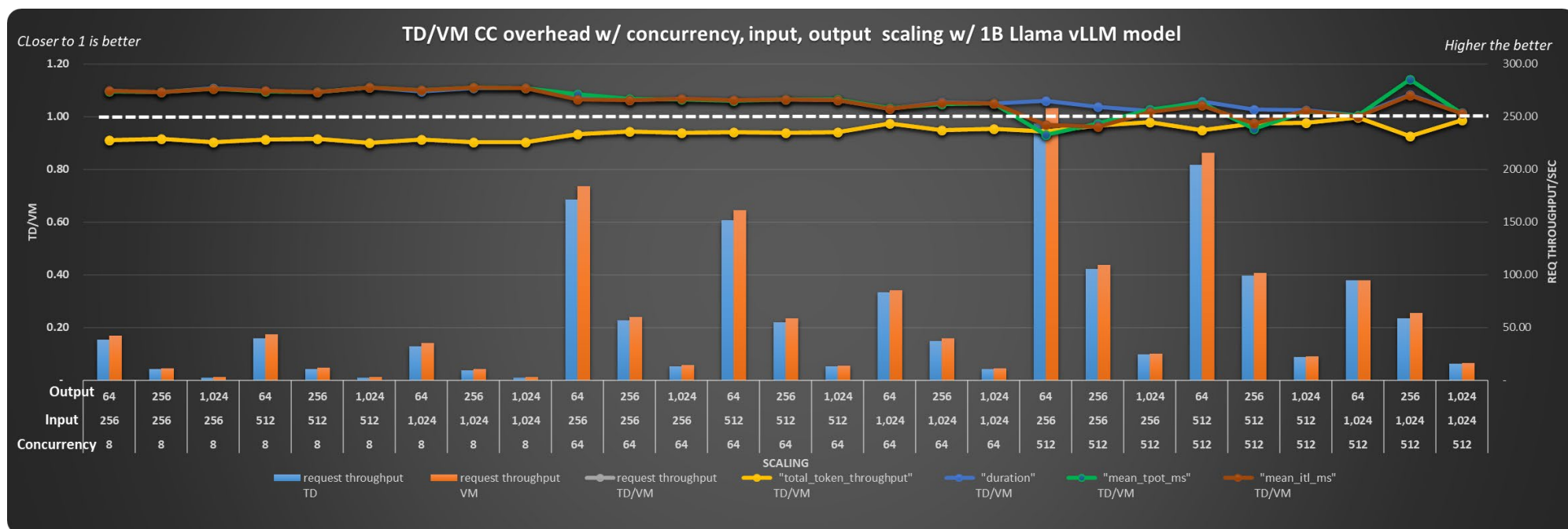


Figure 16: TD/VM vLLM concurrency, input and output length scaling comparison

Additionally, to understand workload and GPU behavior better, we conducted Nsight Systems profiling for a subset of these configurations across TD and VM configurations. Let's investigate these individually:

Disclaimer: In non-profiled execution, CC overhead remains approximately flat across the evaluated input/output scaling points for the fixed batch size of 8, indicating that the end-to-end application-visible overhead is relatively stable in this regime. However, Nsight-based profiling reveals that the relative contribution of CC-sensitive CPU-GPU interaction overheads decreases as workload execution time increases. Thus, profiling is useful for exposing the underlying overhead composition, even when the aggregate end-to-end overhead trend is weak or masked by runtime noise.

7.3.1 Concurrency scaling

To highlight concurrency scaling behavior, selected data points from the larger graph in Figure 16 are shown below in Figure 17, showing that the relative TD/VM overhead decreases with increasing concurrency.

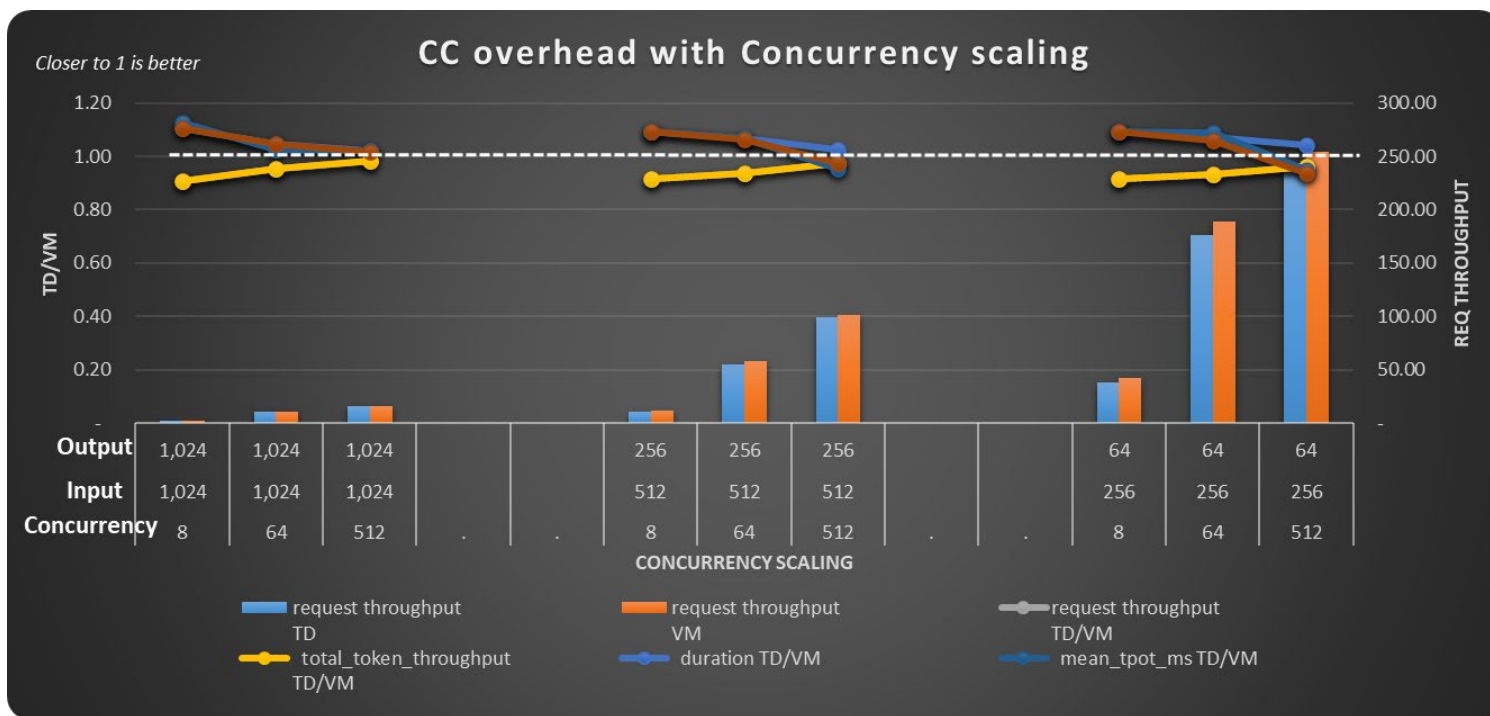


Figure 17: TD/VM vLLM concurrency scaling

To further investigate this trend, Nsight profiling is used to analyze the underlying runtime behavior for 2 configurations- concurrency of 8 vs 64, with input length of 256 and output length of 64.

Disclaimer: Profiling is used in this section to analyze execution behavior and saturation effects, as instrumentation provides visibility into GPU activity. However, at higher concurrency, the profiling overhead (for example, tracing and synchronization) increases and perturbs the absolute CC overhead measurements. Additionally, experiments were conducted with `max_num_seqs=16`, so higher client-side concurrency (for example, 64 or more) reflects

increased offered load rather than fully admitted execution beyond 16 in-flight sequences. Absolute CC overhead trends are therefore derived from non-instrumented runs, which show improved amortization with increasing concurrency.

7.3.1.1 Workload Characteristics

As max-concurrency parameter increases, the maximum number of concurrent requests executed at a time by the GPU increases, provided sufficient system resources are available (for example, GPU memory headroom and a sufficient number of input prompts num-prompts). Higher concurrency increases the input token rate because multiple sequences are processed in parallel. For the 1B model it also increases overall output tokens generated per second, as corresponding throughput increases with concurrency scaling from 8 to 64, as shown in Figure 18. It also helps reduce overall workload run duration to ~0.5x, as it processes the total prompts defined by num-prompts faster due to higher concurrency. Both TTFT and TPOT latencies increase by a different extent. Note that this does not change the total input tokens processed or output tokens generated.

Under confidential computing, an increase in concurrency seems to improve CC performance overhead as measured by benchmark duration, output tokens per second, requests per second, TTFT, and TPOT.

7.3.1.2 Kernel execution

Profiling for concurrency of 8 vs 64, with input length of 256 and output length of 64 shows that, while the number of kernels executing on the GPU decreases to 0.5x with an increase in concurrency, the total duration of hotspot kernel execution increases by >2x, likely because the GPU is now processing higher batch dimension (for example, for `nvjet_tst_384x8_64x4_2x1_v_bz_TNT` function compared to the `nvjet_tst_384x16_64x4_2x1_v_bz_TNT` function). Confidential computing is observed not to change the duration or count of these kernels as expected.

7.3.1.3 Kernel/Graph launches

In line with above observations, the number of kernels and graphs launched as measured by `cudaLaunchKernel` and `cudaGraphLaunch_v10000` reduces to 0.5x with concurrency scaling. This reduces total kernel launch times to 0.5x with an increase in concurrency. Confidential computing increases kernel-launch duration by 1.5x to 2x due to the reasons explained earlier.

7.3.1.4 Memory bandwidth

Host to device traffic as measured by *CUDA Memcpy Device-to-Device* increases ~2x as expected, since 2x concurrent requests are being processed simultaneously, Device to Host memory bandwidth increases only marginally. Confidential computing reduces these bandwidths to nearly half, due to its overheads, which means latency of memory operations are doubled.

In summary, workload-level performance shows that the increase in kernel/graph launch latency and H2D/D2H transfer durations introduced by confidential computing have a diminishing impact on overall throughput as concurrency scales. With higher concurrency, the workload spends proportionally more time in GPU kernel execution and less time in control-path and data-movement overhead. This suggests that the increase in kernel execution duration from larger batches will likely outweigh the additional latency introduced by confidential computing in launch and data transfer paths,

making these overheads less dominant in the overall runtime. As shown in Figure 18, CC overhead reduces from 0.87x to 0.89x when concurrency is scaled from 6 to 64 as measured by *Requestthroughput(req/s)* and *Outputtokenthroughput(tok/s)*.

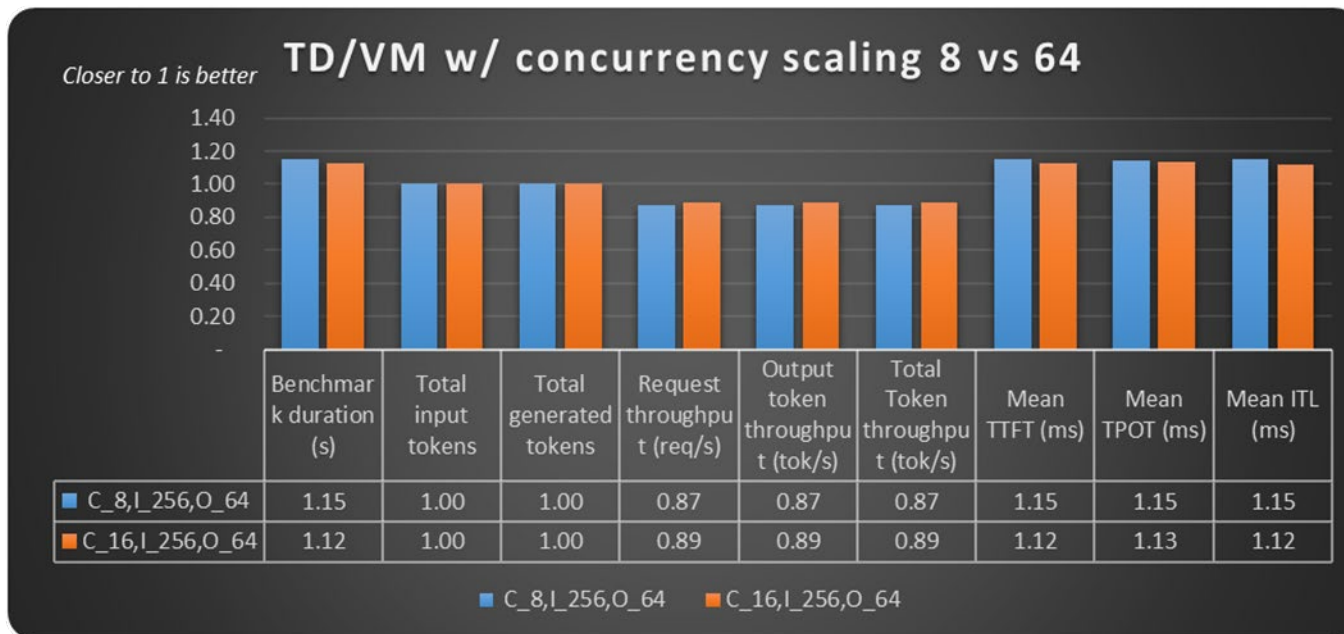


Figure 18: TD/VM workload metrics ratio with Concurrency scaling with Nsights profiling

7.3.2 Input Length scaling

To understand input length scaling behavior, selected data points from Figure 16 are shown below in Figure 19. This shows that scaling input length does not always result in a clearly distinguishable reduction in CC performance overhead.

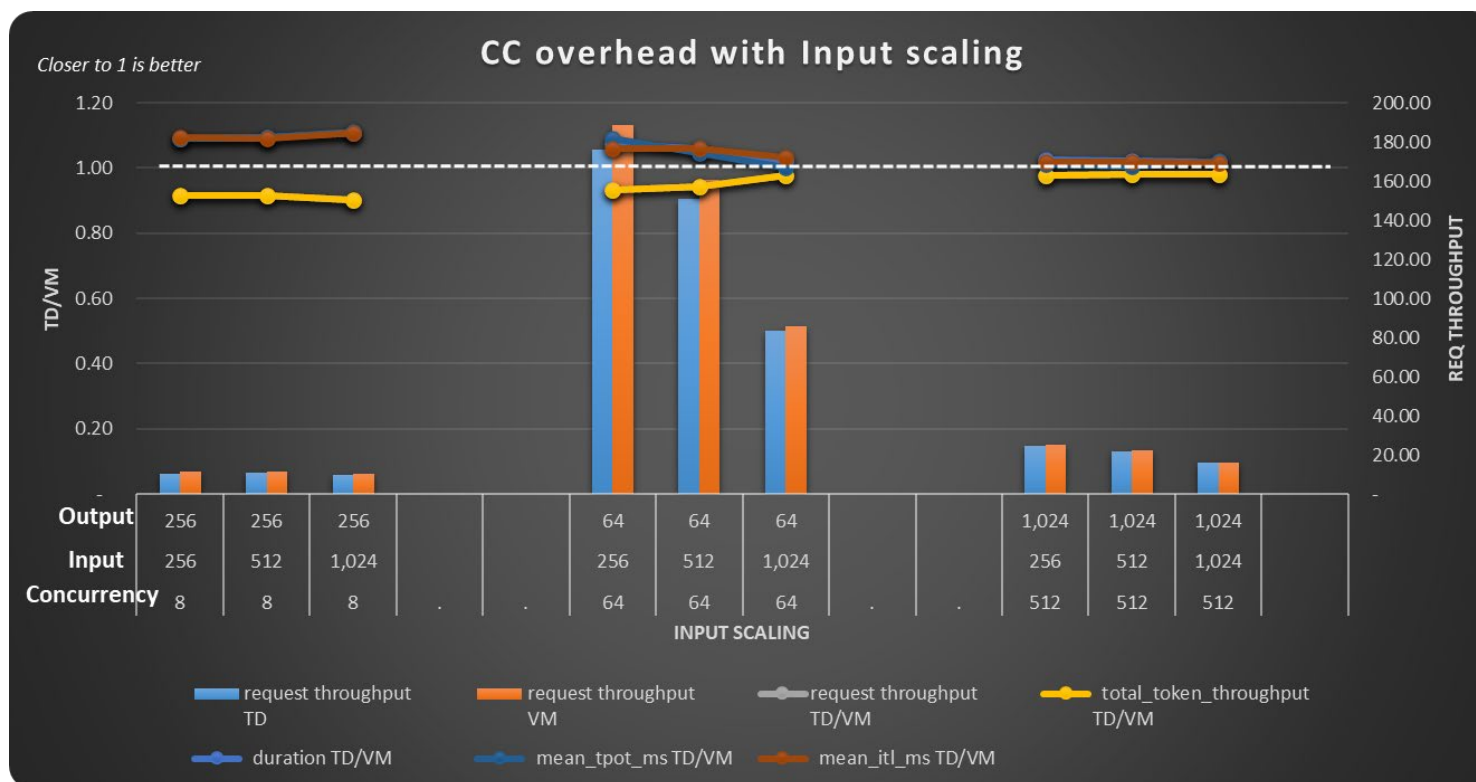


Figure 19: TD/VM vLLM input scaling.

To further investigate this trend, Nsight Systems profiling is used to analyze the underlying runtime behavior for 2 configurations- input length scaled from 256 vs 1024, with concurrency of 8 and output length of 64.

7.3.2.1 Workload characteristics:

Increasing the input length increases the number of input tokens processed per request. For example, when input length is increased from 256 to 1024 tokens, the total number of input tokens processed rises from approximately 100,000 to 400,000, as reported by the workload. Correspondingly, the input token rate increases by approximately 3.5x, reflecting the higher volume of prompt tokens processed during the prefill phase. However, both the total number of output tokens generated and the output token rate decrease. This is due to a reduction in request throughput: longer prompts increase per-request compute time, reducing the number of requests that can be served within the same time window.

Both input and output token rates are ultimately governed by how efficiently the CPU and GPU can process requests, including control-path handling, data movement, and GPU execution efficiency, as discussed in the following sections.

For instance, when input is scaled from 256 to 1024, TTFT increased 2x, while TPOT increases marginally. Increase in input tokens are likely to lengthen the prefill phase of the LLM.

GPU SM utilization is observed to increase by a couple of percentage points, while no distinct difference is observed in KV cache utilization due to increase in input length

With confidential computing, increasing input length improves the CC overhead from 0.87x to 0.91x for throughput and 1.11x for TPOT latencies and benchmark duration, as shown in Figure 20.

7.3.2.2 Kernel Execution

Profiling the increase in input length from 256 vs 1024, with concurrency of 8 and output length of 64, shows that, while the number of kernels executing on the GPU only increases by a few percent, total execution duration of hotspot kernels increases by between 2x to 120x, likely because GPU now has a longer prefill duration as measured by `cutlass::device_kernel<flash::enable_sm, nvjet_tst_384x8_64x4_2x1_v_bz_TNT`, and `nvjet_tst_192x208_64x4_2x1_v_bz_coopB_TNT` functions. Note that function hotspots are observed to change as concurrency increases. As expected, confidential computing does not change the duration or count of these kernels.

7.3.2.3 Kernel Launches

With an increase in input length, the number of kernel and graph launches measured by `cudaLaunchKernel` and `cudaGraphLaunch_v10000` decreases, which in turn reduces the overall time spent on kernel launches. With confidential computing, both kernel and graph launch durations are seen to increase by 1.5x to 2x, as expected, since the volume of data moving across the CPU and GPU per unit of request has increased, thereby increasing the CC overhead of encryption/decryption/integrity as explained before.

7.3.2.4 Memory bandwidth

As expected, bandwidth of data transferred from the CPU to the GPU as measured by `memcpy_dev_to_dev` increases by ~1.4x along with an increase in input length. Please note that changing a workload parameter can change the overall GPU and/or CPU performance, which can impact overall workload throughput. This in turn influences how many requests can be sent to the GPU. Hence, we may not see a linear 4x increase in data bandwidth along with a 4x increase in input length size. An increase in kernel execution duration leads to an overall decrease in requests processed, which leads to a decrease in bytes sent from device to host, which are likely output tokens.

With confidential computing, while the number of transfers between host and device remains roughly the same, the overall duration of memory operations increases ~2x, as expected, due to the security-related operations previously discussed.

In summary, workload-level performance indicates that input length scaling, which increases kernel execution duration by approximately 2x, reduces the relative impact of confidential computing overhead. Although kernel launch latency and CPU–GPU memory transfer durations increase under confidential computing, these costs become less dominant as they are amortized over longer GPU execution times. This suggests that the increase in execution duration likely outweighs the additional latency introduced by confidential computing in launch and data transfer paths, leading to a proportionally reduced

performance impact with input length scaling. Precise mathematical derivation is not attempted here. An increase in overall kernel execution duration leads to a longer time spent by the GPU processing a request, which reduces CC performance impact, as expected. This is shown in Figure 20 below.

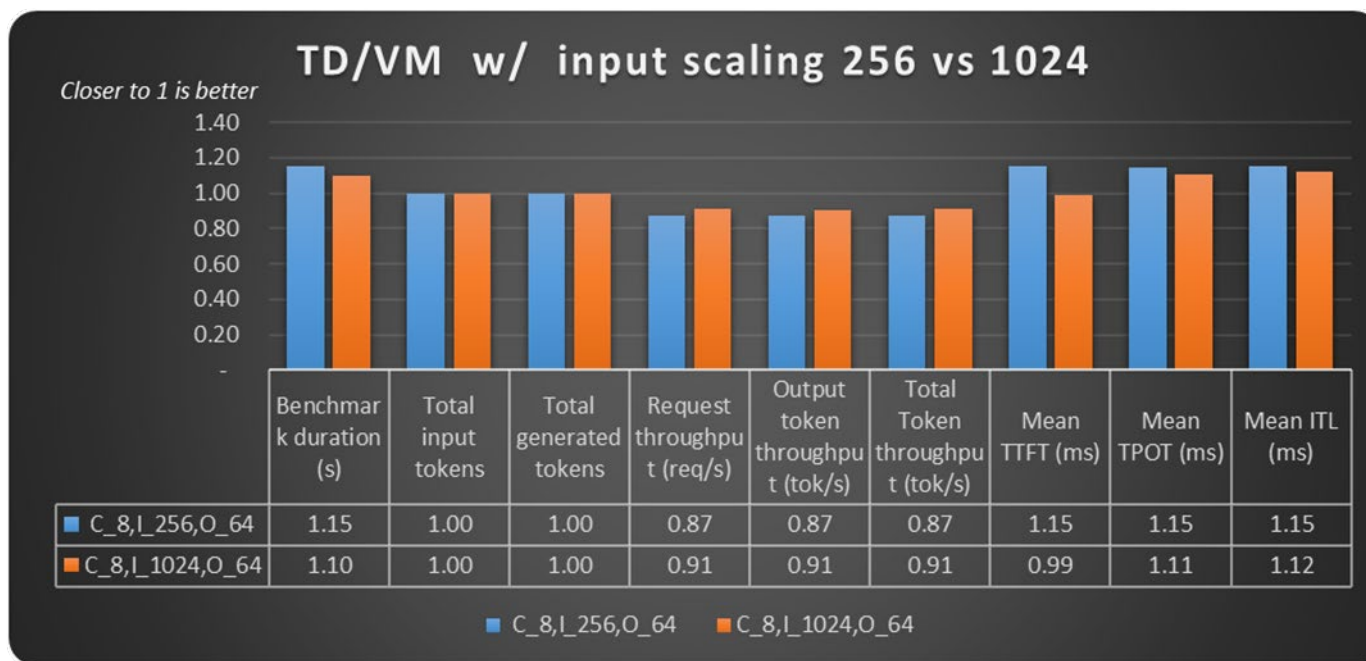


Figure 20: TD/VM workload metrics ratio with input length scaling

7.3.3 Output Length scaling

To understand output length scaling behavior, selected data points from Figure 16 are shown below in Figure 21. It shows that scaling output length does not always clearly reduce CC performance overhead.

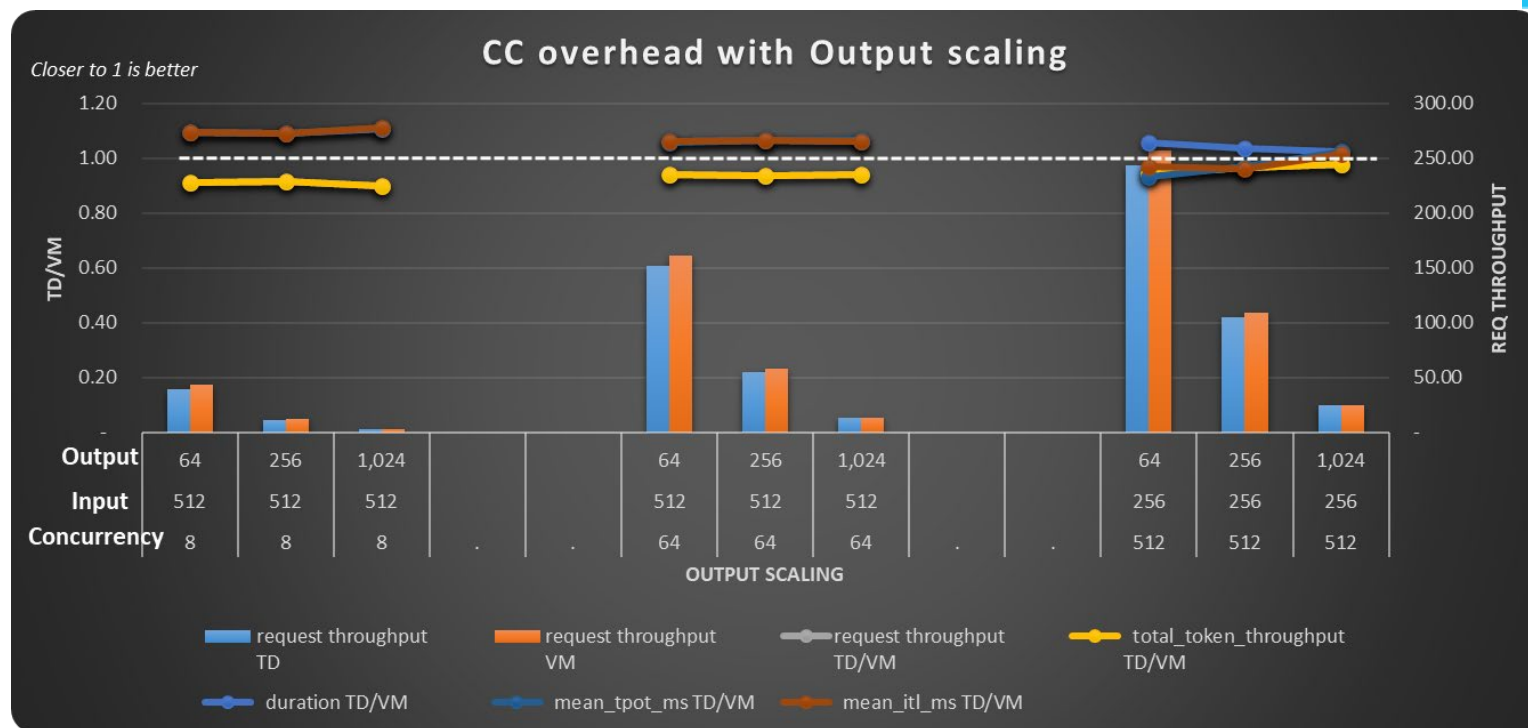


Figure 21: TD/VM vLLM output scaling

7.3.3.1 Workload Characteristics

Scaling output length increases the number of output tokens generated per request during the decode phase. For example, when output length is increased from 64 to 1024 tokens, total output tokens generated increases by approximately 16x, while total input tokens remain unchanged. Since output generation occurs during the decode phase, increasing output length primarily extends the decode duration. In the profiled case, this results in roughly a 15x reduction in request throughput and a corresponding ~15x increase in total workload execution duration. As expected, the input token rate decreases due to fewer requests being processed per unit time.

Output tokens per second increase only marginally, while TTFT and TPOT remain largely unchanged. This suggests that the per-token decode efficiency is stable and that the GPU has sufficient compute and memory headroom to sustain steady-state generation performance.

Supporting this, SM utilization increases only slightly (by a few percentage points, reaching ~70%), and GPU KV-cache usage increases by less than 1%. These observations indicate that the workload remains within available resource limits, even at higher output lengths.

With confidential computing, output length scaling improves the CC overhead from 0.87x to 0.89x for throughput, and from 1.15x to 1.11x for TPOT latencies and benchmark duration, as shown in Figure 22.

7.3.3.2 Kernel Execution

Profiling output length scaling from 64 vs 1024, with concurrency of 8 and input length of 256, shows that, both the number and total duration of kernels executing on the GPU increases 9x as measured by the `nvjet_tst_384x8_64x4_2x1_v_bz_TNT` and `vllm:reshape_cache_flash` functions, likely because the GPU is now experiencing a longer decode duration. Confidential computing does not make any significant difference to the count or duration of kernel execution, as expected.

7.3.3.3 Kernel Launches

With an increase in output length, the number of kernel and graph launches measured by `cudaLaunchKernel` and `cudaGraphLaunch_v10000` increase by 8x-9x, which shows a corresponding increase in total kernel launch duration.

With confidential computing, both kernel and graph launch durations are seen to increase by ~ 2x, as expected, since the volume of data moving across the CPU and GPU per unit of request has increased, thereby increasing the CC overhead of encryption/decryption/integrity as explained before.

7.3.3.4 Memory bandwidth

As expected, bandwidth of data transferred from the CPU to the GPU as measured by *CUDA Device-to-Device* decreases by 0.9x with increase in output length due to a reduction in the overall request rate caused by the increase in kernel execution time. Changes in bandwidth from device to host as measured by *CUDA Memcpy Device-to-Host* is not significant.

As expected, with confidential computing, while the number of transfers between host and device remains roughly the same, the overall duration increases by up to 2.7x due to the security related operations previously discussed.

In summary, workload-level profiling shows that output length scaling significantly increases GPU kernel execution duration, while confidential computing does not materially impact kernel execution itself. Although kernel/graph launch durations and CPU–GPU transfer times increase under confidential computing, these overheads are outweighed by the substantially longer execution phases at higher output lengths. As a result, the relative impact of confidential computing on overall performance is reduced, suggesting that increased execution time from longer decode phases dominates the additional latency introduced in launch and data transfer paths. Figure 22 shows a reduction in CC overhead from 0.87x to 0.89x with output scaling from 64 to 1024.

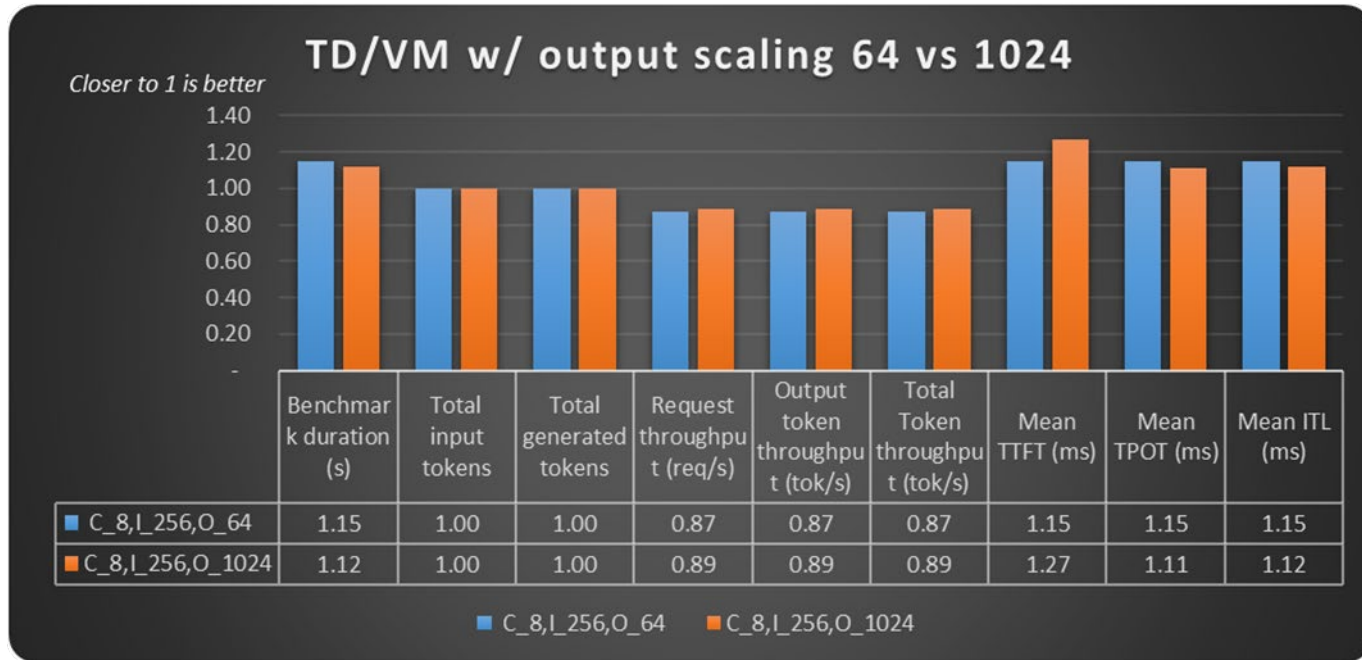


Figure 22: TD/VM workload metrics ratio with output length scaling with Nsight Systems profiling

8.0 Optimization and Mitigation Summary

The performance impact of confidential computing arises from architectural transition overheads, encrypted CPU–GPU communication, and additional page handling in virtualized environments. These overheads propagate into increased kernel launch latency, data transfer duration, and synchronization costs. Based on our layered analysis, optimization strategies can be organized across four architectural levels.

- At the application level, increasing compute intensity through higher concurrency, larger batch sizes, and CUDA Graph capture helps amortize fixed encryption and launch overheads. Reducing frequent host synchronization and minimizing small, repeated H2D/D2H transfers further limits control-path amplification.
- At the system level, aligning CPU cores and memory locality with the GPU device reduces scheduling and NUMA-related inefficiencies. CPU pinning, isolation, and reducing scheduler-induced IPIs help mitigate exit amplification effects.
- At the Intel TDX and virtualization layer, minimizing private-to-shared page conversions, reducing CPA flush events, lowering Intel TDX transition frequency, and enabling posted interrupts reduce both exit rate and exit duration. Future mechanisms such as Intel TDX Connect can structurally eliminate bounce-buffer overhead and further reduce shared-memory transitions.
- At the GPU and driver layer, improving CPU-side cryptographic scalability (for example, leveraging Intel AVX-512 VAES efficiently and parallelizing encryption across cores) alleviates host-side bottlenecks during launch and data transfer. Increasing compute dominance through kernel fusion or workload scaling further amortizes fixed confidential computing overheads.

Overall, the most effective system-level strategy is not merely reducing overhead events, but shifting workloads toward compute-dominated execution, where fixed control-path and encryption costs become proportionally less significant. This principle explains the observed reduction in confidential computing impact under larger models, higher concurrency, and longer input lengths.

Table 2: Confidential Computing Overheads Optimization Summary

Tier	Factor	Optimization	Goal
Tier 1	Page conversions	Reduce shared-to-private page conversions during run time by pre-establishing a pool of DMA-accessible shared pages.	Minimize cpa_flush, x2APIC-ICR-related VM exits and guest TLB shootdowns.
Tier 1	IPI overhead	Reduce IPI overhead by CPU isolation of the workload. For example, using the <code>nohz_full=5-7</code> or <code>rcu_nocbs=5-7</code> flags.	Reduce scheduler related IPIs and potentially reduce TLB shutdown spread.
Tier 1	CPU parallelism for crypto	Modify UVM to use multiple cores for encryption. Ensure Intel AVX-512 VAES is enabled with the latest OpenSSL.	To avoid single-core bottleneck with workloads like NVBandwidth.
Tier 1	NUMA aware	Ensure collocation of VM's resources like vCPU and its memory, with PCI-e GPU device.	Eliminate UPI-encryption related overhead.

Tier	Factor	Optimization	Goal
Tier 1	Reduce Intel TDX transitions	Use polling-mode I/O threads, reduce timer ticks, and reduce shared/private conversions.	Reduce performance overhead due to Intel TDX transitions.
Tier 1	Enable IPI virtualization.	Enable IPI virtualization to reduce performance overhead of VMM exits from ICR register writes	Reduce performance overhead due to Intel TDX transitions from ICR register writes.
Tier 1	Intel TDX Connect	Adopt Intel TDX Connect in the future.	To eliminate most of the performance overheads mentioned in Tier 1.
Tier 1	Reduce UVM page faults	Pre-touch device memory, pre-migrate UVM managed memory to device, and avoid demand paging.	Eliminate GPU-CPU security-related overheads on page migrations from performance critical path.
Tier 1	Minimize CPU-GPU transfers	Increase batch size and combine multiple small data copies.	Eliminate GPU-CPU security-related overheads.
Tier 2	Reduce Sync sensitivity	Avoid excessive <code>cuStreamSynchronize</code> and prefer event-based synchronization.	Reduce CPU/GPU cycles wastage over synchronizations as observed with <code>cuStreamSynchronize</code> and <code>spinKernelDevice</code> with NVBandwidth and <code>cuMemcpyAsync</code> with vLLMs.
Tier 2	Front load all control path	Move every possible control-path activity to serving initialization like model load, CUDA graph capture, or warm up passes.	Eliminates CC overhead due to kernel/graph launch.
Tier 2	Use CUDA Graphs	Use CUDA graphs instead of several small kernels (this is already observed to be in place).	Reduce repeated security costs in the control path from reduced kernel/graph launches.
Tier 2	Increase GPU compute intensity	Larger models and longer kernels reduce CC overhead. Increase per-launch work, for example, by fusing kernels or using large batch size.	Increase execution times by executing larger kernels/graphs to reduce CC overhead due to CPU-GPU data transfers per launch.
Tier 2	Increase stream overlap	Overlap kernel and data transfers whenever possible by using separate streams.	Reduces impact of CC overhead due to data security operations on kernel launch operations.

Tier	Factor	Optimization	Goal
Tier 3	Increase Concurrency	Increase batch size of input prompts.	Increase GPU utilization, and hence kernel execution times (as observed from Nsight Systems), which reduces CC overhead.
Tier 3	Input length scaling	Increase number of input tokens.	Longer prefill duration due to longer input length helps reduce CC overhead.
Tier 3	Output length scaling	Increase number of output tokens.	Longer Decode duration due to longer output length helps reduce CC overhead.
Tier 3	KV cache headroom	Ensure sufficient HBM headroom.	Increasing KV cache headroom enables higher concurrency and improved batching efficiency, which increases compute dominance and amortizes fixed confidential computing overhead.

9.0 Summary

This paper presented a cross-layer analysis of confidential computing performance for GPU-accelerated AI workloads using Intel TDX and NVIDIA H100 GPUs. Because NVIDIA GPUs and drivers are treated as third-party components, our analysis is limited to externally observable system-level behavior and avoids attribution to internal GPU hardware or firmware mechanisms. By examining NVBandwidth microbenchmarks and vLLM serving workloads across multiple model sizes and scaling configurations, we systematically decomposed confidential computing performance impact into Tier-1 architectural causes, Tier-2 runtime propagation effects, and Tier-3 workload-level manifestations.

Our analysis shows that confidential computing performance overhead is primarily concentrated in control-path and data-movement operations rather than GPU compute execution. Performance of kernel execution within the GPU's trusted memory hierarchy remains unaffected. The dominant contributors to overhead include encrypted kernel and graph launch commands, AES-GCM-protected CPU-GPU data transfers, Intel TDX transition latency, and private-to-shared page handling. These architectural effects propagate into increased launch latency, extended memory transfer durations, and amplified synchronization and queuing delays under certain workloads.

Importantly, we observe that confidential computing overhead diminishes as workloads become more compute dominated. Larger models, increased concurrency, and longer input sequences shift execution time toward sustained GPU computation, amortizing fixed encryption and transition costs. As a result, while smaller or launch-intensive workloads may experience double-digit percentage degradation, large-scale LLM inference workloads exhibit only modest performance impact.

The findings demonstrate that confidential AI inference is practical and scalable on modern hardware platforms. With appropriate system tuning and workload configuration, the security protections of confidential computing can be achieved with limited and predictable performance overhead. As hardware and virtualization mechanisms continue to evolve - particularly with enhancements such as reduced transition cost and improved device trust models - the gap between confidential and non-confidential execution is expected to narrow further.

10.0 *References*

1. [Confidential Compute on NVIDIA Hopper H100](#)
2. [Dissecting Performance Overheads of Confidential Computing on GPU-based Systems](#)
3. [Performance Considerations of Intel Trust Domain Extensions on 4th generation Intel Xeon Scalable Processors](#)
4. [Runtime Encryption of Memory with Intel® Total Memory Encryption–Multi-Key \(Intel® TME-MK\)](#)
5. [Intel TDX Architecture specification](#)
6. [Intel® TDX Module Interrupt Virtualization Architecture Specification](#)
7. [Timers: Linux Kernel](#)
8. [Intel TDX Connect Architecture Specification](#)
9. [In-Depth Analyses of Unified Virtual Memory System for GPU Accelerated Computing](#)
10. [NVIDIA GPU Confidential computing demystified](#)

11.0 Notices and Disclaimers

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document; however, the information reported herein is available for use in connection with the mitigation of the security vulnerabilities described.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

Intel technologies may require enabled hardware, software or service activation.

No product or component can be absolutely secure.

Your costs and results may vary.

Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not "commercial" names and not intended to function as trademarks.

Performance varies by use, configuration and other factors. Learn more on the Performance Index site. Real world performance impacts may vary, and any stated figures herein are estimates.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

11.1 Configurations

For workloads and configurations, see the following resources. Results may vary.

Test by Intel as of 19 March 2026

1-node, Intel Corporation BeechnutCity, 2x Intel® Xeon® 6787P processors with 86 cores, 350W TDP, Intel® Hyper-Threading Technology (Intel® HT Technology) On, Turbo On, Total Memory 256GB (16x16GB DDR5 6400MT/s [6400MT/s]), BIOS BHSDCRB1.IPC.3544.P25.2411260715, microcode 0x10003a2, 1x I210 Gigabit Network Connection, 1x 894.3G Micron_7450_MTFDKBG960TFR, 6x 931.5G INTEL SSDPE2KX010T7, C states enabled, Max frequency 3.8 GHz, 4 NUMA nodes, 2MB Transparent huge pages with madvise config in Host OS, Balanced Performance Bias, Scaling governor is powersave, Energy performance preference is Performance, Ubuntu* 25.04, 6.14.0-1004-intel, gcc version gcc 14.2.0, virsh libvirt version: 11.0.0.

Memory Encryption (TME) Enabled, Total Memory Encryption Multi-Tenant (TME-MT) Enabled, Memory Integrity (Disabled), UPI FLE request Disabled, Cryptographic Algorithm (AES-XTS256), Trust Domain Extensions (TDX) Enabled, TTME-MT Keys 63, TDX keys 64, TME-MT/TDX key split 1, TED Secure Arbitration Mode Loaded (SEAM Loader) Enabled, Trust Domain Extensions Connect (TDX Connect) Disabled, SGX enabled.

NVIDIA H100 GH100, NVIDIA driver version 570.148.08, UVM driver version 570.148.08, GPU CUDA version 12.8, VBIOS Version 96.00.5E.00.01, GSP Firmware Version 570.148.08

Guest 8vCPUs pinned to individual cores 87-94 for a TD, and 96-103 for a VM, 32GB of memory, QEMU emulator version 9.2.1 (Debian 1 9.2.1+ds-1ubuntu4+tdx2.0~ppa2), OpenSSL version OpenSSL 3.0.13, Ubuntu 24.04.2 LTS, Kernel 6.8.0-60-generic.

vLLM version 0.9.1, nvbandwidth version v0.8.

11.2 Workload and Tracing Commands

11.2.1 Model size scaling:

Model names: Llama-3.2-1B, Llama-3.2-3B, Llama-3.2-11B-Vision-Instruct, internlm2_5-20b-llamafied, llama-30b-instruct, Llama-3.3-70B-Instruct-quantized.w8a8

Start the vllm serving engine:

```
Llama-3.3-70B-Instruct-quantized.w8a8: #vllm serve /mnt/data2/repos/$1/ --disable-log-requests --max_num_seqs 16 -
--max_model_len 4096 --kv-cache-dtype fp8 --gpu-memory-utilization 0.98
llama-30b-instruct : #vllm serve /mnt/data2/repos/$1/ --disable-log-requests --max_num_seqs 16 --max_model_len 2048
others: #vllm serve /mnt/data2/repos/$1/ --disable-log-requests --max_num_seqs 16 --max_model_len 4096
```

vllm benchmark run:

```
#python benchmark_serving.py --backend vllm --model /mnt/data2/repos/$1/ --endpoint /v1/completions --dataset-name
sharegpt --dataset-path ShareGPT_V3_unfiltered_cleaned_split.json --num-prompts 500
```

11.2.2 Concurrency+ Input + Output scaling

```
#conc=16;
#in_len=256;
#out_len=64;
#tag="c${conc}_in${in_len}_out${out_len}"
#vllm serve /mnt/data2/repos/Llama-3.2-1B --disable-log-requests --max_num_seqs 64 --max_model_len 2048
#vllm bench serve --model /mnt/data2/repos/Llama-3.2-1B --base-url "http://127.0.0.1:8000" --dataset-name random -
-random-input-len "$in_len" --random-output-len "$out_len" --num-prompts 400 --max-concurrency "$conc" --seed 123
--save-result --result-filename "/root/Analysis/data_Nsights_feb18/Nsights_batch_scaling/${tag}.json"
```

11.2.3 Nsight Systems tracing

```
#vllm serve /mnt/data2/repos/Llama-3.2-1B --disable-log-requests --max_num_seqs 16 --max_model_len 2048
#nsys profile -o <file_location> --trace=cuda,nvtx,osrt < command>
```

11.2.4 NVBandwidth CE test

```
#!/root/nvbandwidth/nvbandwidth -t 0
#!/root/nvbandwidth/nvbandwidth -t 1
```

11.2.5 Enable CC mode in GPU

```
#!/home/ubuntu/tdx/guest-tools/nvtrust/host_tools/python/nvidia_gpu_tools.py --set-cc-mode=on --reset-after-cc-
mode-switch --gpu-bdf=0000:b8:00.0
#!/home/ubuntu/tdx/guest-tools/nvtrust/host_tools/python/nvidia_gpu_tools.py --set-ppcie-mode=off --reset-after-
ppcie-mode-switch --gpu-bdf=0000:b8:00.0
```

11.2.6 Disable CC mode in GPU

```
#!/home/ubuntu/tdx/guest-tools/nvtrust/host_tools/python/nvidia_gpu_tools.py --set-cc-mode=off --reset-after-cc-
mode-switch --gpu-bdf=0000:b8:00.0
#!/home/ubuntu/tdx/guest-tools/nvtrust/host_tools/python/nvidia_gpu_tools.py --set-ppcie-mode=off --reset-after-
ppcie-mode-switch --gpu-bdf=0000:b8:00.0
```

11.2.7 Enable CC debug mode in GPU (for Nsight Systems profiling)

```
#!/home/ubuntu/tdx/guest-tools/nvtrust/host_tools/python/nvidia_gpu_tools.py --set-cc-mode=devtools --reset-after-
cc-mode-switch --gpu-bdf=0000:b8:00.0
# /home/ubuntu/tdx/guest-tools/nvtrust/host_tools/python/nvidia_gpu_tools.py --set-ppcie-mode=off --reset-after-
ppcie-mode-switch --gpu-bdf=0000:b8:00.0
```