Dissecting Performance Overheads of Confidential Computing on GPU-based Systems

Yang Yang
University of Virginia
Charlottesville, Virginia, USA
yangyang@virginia.edu

Mohammad Sonji
University of Virginia
Charlottesville, Virginia, USA
npv2tk@virginia.edu

Adwait Jog
University of Virginia
Charlottesville, Virginia, USA
ajog@virginia.edu

***Abstract*—Confidential computing (CC) is a critical technology for protecting data in use. By leveraging encryption and virtual machine (VM) level isolation, CC allows existing code to run without modification while offering confidentiality and integrity guarantees. However, the performance impact of CC in GPU-based systems can be significant. In this work, we present a comprehensive performance evaluation of CC guided by a simple performance model. Specifically, we start by evaluating CUDA applications with a focus on data transfer, memory management, encryption, kernel launch, and kernel execution. We also present a detailed event-level analysis of these applications, revealing that the execution times of kernels that do not use unified virtual memory (UVM) are mostly unaffected, while associated kernel launch overhead and queuing time increase significantly. On the other hand, the execution time of kernels using UVM increases drastically under CC, in addition to other launch and queuing overheads. We also study CNN training and LLM inference to see how CC overhead would affect them. Finally, we consider several optimization techniques, including kernel fusion, overlapping, and quantization, towards addressing the overheads of CC.**

***Index Terms*—Confidential Computing, GPUs, Performance Evaluation**

I. Introduction

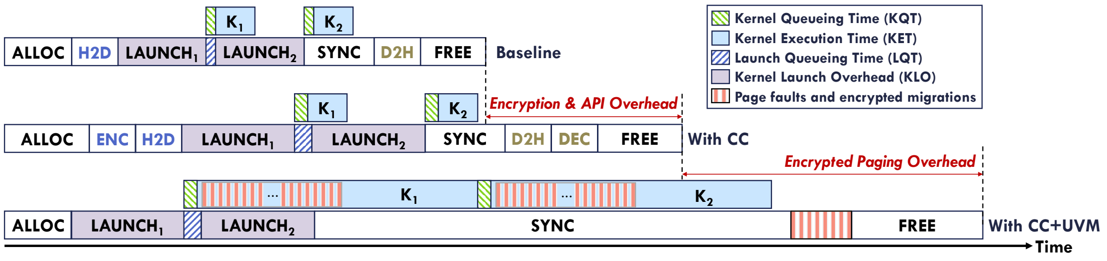
Graphics Processing Units (GPUs) have become the leading accelerators due to their ability to meet the increasing computational demands of diverse workloads across various computing domains. In recent years, general-purpose GPUs (GPGPUs) have extended their support beyond traditional graphics workloads to more general tasks, such as deep learning, graph processing, and scientific computation. However, the deployment of GPUs for services in the cloud has raised concerns about data privacy and security, as users may be required to upload potentially sensitive information. With an increasing volume of sensitive data being processed remotely, the demand for privacy and security has grown rapidly. This demand has reinforced the need for confidential computing [1]–[3], which ensures data confidentiality and integrity through hardware-enabled Trusted Execution Environments (TEEs) [4]–[11].

NVIDIA recently introduced its confidential computing (CC) solution for H100 GPUs [3], [12]–[14], marking the first commercialized GPU TEE product. H100 CC is built on virtual machine (VM)-level CPU TEEs, such as Intel Trust Domain Extensions (TDX) or AMD Secure Encrypted Virtualization-Secure Nested Paging (SEV-SNP).[1] Contrary to Intel Software Guard Extensions (SGX) [11], [15], a key advantage of this VM-level isolation is that it requires no code changes when transitioning to CC. For example, the device driver and OS can still run inside such a virtual machine (called Confidential Virtual Machine (CVM) in AMD terminology, or Trust Domain (TD) in Intel terminology), which was difficult to achieve in SGX. However, overhead arises from both CPU [16], [17] and GPU-specific TEEs [3], [18], [19]. Previous studies [18] have evaluated the overhead of deploying large language model (LLM) inference on H100 CC systems. However, the reasons behind this overhead, i.e., why GPU application performance degrades under CC, have not been investigated in depth. Without understanding the low-level breakdown of such overheads, mitigation strategies remain difficult to explore. In this paper, our goal is to better understand the overhead of CC in GPU-based systems and explore potential mitigation strategies.

To this end, we first propose a simple performance model (Sec. V) to better understand the GPU performance. This model identifies key metrics (i.e., data copy, launch, queuing, and kernel execution) affecting GPU application performance and demonstrates how these metrics contribute to end-to-end performance. Using this model, we analyze data transfer, memory allocation, and CC-required encryption overheads (Sec. VI-A), uncovering memory management properties that are specific to CC. We conduct a detailed event-level analysis of kernel launch and execution activities (Sec. VI-B), revealing that kernel execution time (KET) remains unaffected when unified virtual memory (UVM) is not used (i.e., in non-UVM scenarios). However, this is not the case when UVM is used, as it incurs significant overhead. Also, kernel launch overhead (KLO), launch queuing time (LQT), and kernel queuing time (KQT) are significantly affected.

An overview of the overhead breakdown is presented in Fig. 1. Memory allocation and de-allocation take more time due to TDX's memory management design. Data copying incurs additional overhead due to encryption and decryption (only CPU side overheads are shown), kernel launches take longer, and queuing times increase for both kernel execution and launches. While non-UVM kernels perform nearly the same regarding KETs, UVM kernels experience extremely high overhead due to encrypted paging. This is further detailed in Sec. VI. Using customized microbenchmarks (Sec. VII-A),

[1]We use Intel and NVIDIA terminologies in this study.



Kernel Queueing Time (KQT)
Kernel Execution Time (KET)
Launch Queueing Time (LQT)
Kernel Launch Overhead (KLO)
Page faults and encrypted migrations
K1
K2
ALLOC
H2D
LAUNCH1
LAUNCH2
SYNC
D2H
FREE
Baseline
Encryption & API Overhead
ENC
DEC
With CC
Encrypted Paging Overhead
With CC+UVM
Time

Fig. 1: Overview of end-to-end GPU application performance under different confidential computing (CC) settings: baseline execution (top, CC-off), execution with confidential computing (middle, CC-on), execution with confidential computing under unified virtual memory (bottom, CC-on).

we further investigate such kernel launch behaviors and evaluate their impact on the overall performance.

Next, we explore optimization techniques, such as kernel fusion and overlapping, and associated trade-offs (Sec. VII-A). We also evaluate the performance of CNNs and LLMs (Sec. VII-B) and examine the impact of quantization techniques on them. To the best of our knowledge, this is the first work that conducts a detailed characterization of performance overheads related to GPU-based confidential computing and evaluates optimizations to address those overheads. Our contributions are summarized as follows:

- We present a performance model that helps in dissecting key overheads of GPU-based confidential computing into distinct components, including data transfer, encryption, queuing, kernel launch time, and kernel execution.
- We perform an event-level analysis of CUDA applications under both UVM and non-UVM scenarios. With CC, we observe significant increases in kernel launch overheads and queuing times. Moreover, UVM kernels experience substantial slowdowns, while non-UVM kernel execution remains largely unaffected.
- To address CC overheads, we explore several optimizations, including kernel fusion and overlapping. For CNNs and LLMs, we also evaluate quantization techniques and their potential to mitigate CC overheads.

II. Background

In this section, we provide background related to GPU-based confidential computing and unified virtual memory.

A. GPU-based Confidential Computing

Confidential computing (CC) [1], [2], [20] aims to ensure the confidentiality and integrity of data (and code) by leveraging hardware-based Trusted Execution Environments (TEEs). CC protects privacy-critical workloads from a wide range of threats and physical attacks. This includes cases where the underlying infrastructure, such as hypervisors or operating systems, is compromised. This paradigm has gained significant traction as modern applications rely on cloud and heterogeneous environments to meet the computational demands of emerging workloads.

In this paper, we focus on a heterogeneous CC system, which leverages both CPU and GPU [20]. The software architecture and the underlying hardware for confidential computing are shown in Fig. 2. The CPU, which supports Intel's Trust Domain Extensions (TDX) [5], [21]–[23] could provide virtual machine (VM) level isolation and encrypted memory for trust domains (TDs).[2] Unlike Intel SGX [4], [15], [24], which adopts a process- and region-based isolation model, VM-level isolation makes TDs capable of running legacy applications without code modifications. To support TDs, Intel introduced the *Secure Arbitration Mode (SEAM)* [16], [25] as an extension to the Intel Virtual Machine Extensions (VMX). They also provide a trusted signed component called the *TDX Module*, which manages interactions between TDs and the host (including the hypervisor). Similar to the VMX root and non-root modes for traditional VMs, SEAM operates with both root and non-root modes, as shown in Figure 2. *Hypercalls* and *seamcalls* [16], [17] are used to handle communication and context switching between the TD, the TDX Module, and the host. Similar to SGX MEE [24], TD's private memory is encrypted using Intel Total Memory Encryption–Multi-Key (TME-MK) [26], [27], a memory encryption engine (MEE) that resides in the memory controller. However, TME-MK utilizes AES-XTS [28], a counter-less symmetric encryption algorithm, which eliminates the overhead of storing secure metadata. Such a light-weighted design allows TME-MK to protect the entire memory space, unlike SGX, which can only secure a limited memory region and could incur significant overhead [29] from encrypted page swapping.

The GPU architecture includes a PCIe controller for interacting with the CPU through memory-mapped I/O (MMIO) and direct memory access (DMA). Internally, the GPU is equipped with several engines, including the command processor (also known as the channel engine), copy engine, compute engines, and graphics engines (only compute engines are shown in Fig. 2) [30], [31]. The GPU features its own dedicated device memory, with each memory partition comprising L2 cache slices, a memory controller, and high-bandwidth memory (HBM) stacks. With GPU passthrough via VFIO [32], the VM or TD can access the GPU directly, as if it were natively attached over PCIe. Note that the GPU must be switched to CC mode before binding to a TD. When a TD needs to offload computation to the GPU, the device driver within the TD sends commands (e.g., set device,

[2]In this paper, we use the terms TD and CVM interchangeably.

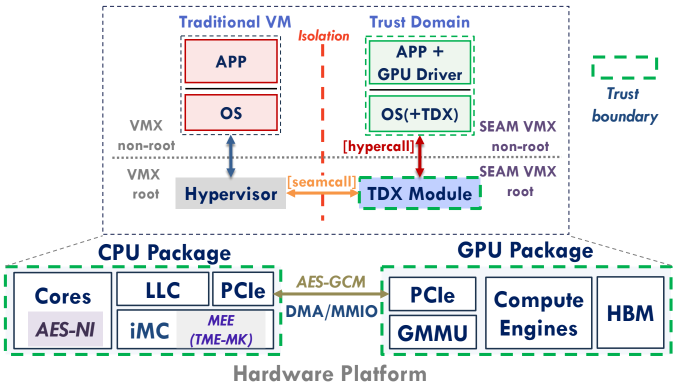


Fig. 2: Architecture overview of CPU-GPU confidential computing. Detailed hardware specifications are documented in Section IV. The dashed green regions are trusted components, including both CPU and GPU packages and the Intel TDX module. Confidential computing must take place in isolated trusted domains (TDs). The GPU is exclusively occupied by either TD or traditional VM. A detailed threat model is described in Sec. III.

kernel launch) to the GPU. This IO request from TD will trigger `tdx_hypercall` to perform a context switch to the TDX Module and then transferred to the host (hypervisor) for handling (i.e., Intel #VE [33]). The command processor [10], [11], [34]–[40] is responsible for handling all commands issued by the device driver. Commands are submitted to MMIO-configurable channels [10], [40] and represent the only pathway for the GPU to receive commands. Each channel is associated with a channel descriptor and page tables [41] and is exclusively linked to a single GPU context.

Once received by the command processor, the commands are distributed to the appropriate engines for execution. For instance, copy engines [30], [36], [37], [42] handle asynchronous memory transfers between the CPU and GPU, while the compute engine, consisting of a collection of general processing clusters (GPCs[3]), executes highly parallel computations in the single-instruction, multiple-thread (SIMT) style. Additionally, the GPU incorporates a GPU Memory Management Unit (GMMU) to manage virtual memory on the GPU, which is further detailed in Sec. II-B. Under confidential computing, CPU-GPU communication over the PCIe bus is encrypted using AES-GCM [3], [43], [44]. This encryption is implemented in software using OpenSSL [45] with `AES-NI` [46] acceleration. Our findings confirm this implementation, and we evaluate the performance of different crypto choices in Sec. VI. Meanwhile, due to TDX isolation requirements and IOMMU limitations, the GPU cannot directly access a TD's private memory. As a result, DMA operations require shared memory managed by the hypervisor, known as a *bounce buffer*, which can be allocated using the `dma_alloc_*` APIs [16], [17], [19], [25], [33], [47]–[49], to temporarily hold the encrypted data being transferred. The Linux kernel `set_memory_decrypted()` can be used to convert a private page into a shared page. Internally, it changes page attributes to bypass TME-MK.[4] It is important to note that GPU HBM itself is not encrypted; the reasoning behind this decision is discussed in Sec. III.

[3]Collection of texture processing clusters (TPCs). Each TPC includes few streaming multiprocessors (SMs).

B. Unified Virtual Memory

Unified Virtual Memory (UVM) [50] enables a shared memory address space between CPU and GPU. By eliminating explicit data copy function calls, UVM simplifies programming for heterogeneous CPU-GPU systems. Using the `cudaMallocManaged` API, the same memory object pointer can be referenced on both the CPU and GPU, enabling data pages to migrate on demand. For example, when the GPU attempts to access a page residing in CPU memory, it triggers a *far fault*. The GPU Memory Management Unit (GMMU) handles this type of page fault request. The request is forwarded to the CPU-side UVM driver [51]. The average latency reported is estimated to take 20 μs - 50 μs [52]–[54]. Subsequently, the requested pages are migrated to the GPU via the PCIe bus. However, naive page migration introduces significant performance overhead compared to the regular *copy-then-execute* model [55]. To mitigate such overhead, NVIDIA GPUs internally employ batching and prefetching techniques [54], [56]. In this paper, we use the term UVM to refer to applications that use the `cudaMallocManaged` API for memory management. In contrast, non-UVM refers to applications that rely on explicit memory transfers using the `cudaMemcpy` API.

III. Threat Model

With Intel TDX and NVIDIA CC, the Trusted Computing Base (TCB) extends beyond the CPU/GPU packages to include the TDX module and firmware on both CPU and GPU sides. As a result, we trust the NVIDIA-provided GPU device driver running inside the TD. Hypervisor [57], [58], CPU memory, and the PCIe interconnect [59], [60] are still considered untrusted and potentially vulnerable to compromise. Although PCIe 5.0 does not natively support Integrity and Data Encryption (IDE) [61]–[63], NVIDIA utilizes Security Protocols and Data Models (SPDM) [43], [64], [65] to attest communication between the CPU and GPU over PCIe. Additionally, AES-GCM is used to encrypt data transferred through the PCIe bus [3], [43]. However, GPU side-channels are still possible under CC [43] (e.g., physical side-channels) and have been explored in other contexts [31], [41], [66], [67].

In GPU-based confidential computing systems, CPU-side DDR memory is considered insecure [68]–[72]. Therefore, memory encryption [24], [73], [74] (i.e., TME-MK) must be applied to preserve the confidentiality and integrity of data residing in TD's private memory. However, HBM in high-end GPUs (e.g., NVIDIA H100 and AMD Instinct MI325X) is assumed to be immune to physical attacks and thus considered to be secure. This assumption [75]–[78] relies on the fact that

[4]Documented in `arch/x86/mm/pat/set_memory.c`.

3D-stacked HBM [79], [80] is fabricated alongside the chip and connected using a silicon interposer. The HBM dies within the same stack are internally connected using through-silicon vias (TSVs). The physical scale is so small that it is unlikely an attacker could intercept or modify the data by inserting a Trojan in the die or interposer [43], [78]. This assumption has been used in previous studies [10], [11], [81]–[83], which also applies to H100 CC, eliminating the need for GPU memory encryption. Although attacks like Rowhammer [84], [85] pose threats to CPU memory, to the best of our knowledge, no real-world Rowhammer attacks have been reported against GPU HBM so far.

IV. Experimental Setup and Methodology

In this section, we describe our evaluation setup. As shown in Table I, the system is built with commercial hardware to support CPU-GPU confidential computing (CC). It features dual Intel Emerald Rapids (EMR) processors (5th Gen Xeon Scalable Processor family [86]) with official TDX support. This setup closely mirrors the architecture depicted in Fig. 2.

TABLE I: Confidential Computing System Setup

Configuration	
CPU	2× 5th Gen Intel Xeon 6530 Gold @2.1GHz, 32 cores
Memory	16× 64GB DDR5 4800MHz (1TB)
TME-MK	Auto bypass enabled
Storage	Micron 5400 PRO 960GB, SATA
System	Supermicro SYS-421GE-TNRT3 (PCIe 5.0)
OS	Ubuntu 22.04.5 LTS (Linux 6.2.0, tdx patched)
Hypervisor	QEMU 7.2.0 (tdx patched)
TDX Tools	TDX 1.5 (tag 2023ww15)
GPU	NVIDIA H100 NVL, 94GB HBM3, PCIe 5.0 ×16 CUDA 12.4, Driver 550.127.05

Each CPU has 32 cores clocked at 2.1 GHz, totaling 64 physical cores across 4 NUMA nodes. The system includes 1 TB of main memory and two NVIDIA H100 NVL 94GB GPUs, with one connected to socket 0 (NUMA node 0) and the other to socket 1 (NUMA node 3). The entire platform is hosted on a Supermicro SYS-421GE-TNRT3 server, which supports both TDX and direct PCIe 5.0 connections. We enabled the TDX feature in the BIOS as suggested by Intel.

The software stack was built following the NVIDIA Confidential Computing Deployment Guide (`v.3.0` [13]). The system runs `Ubuntu 22.04.5 LTS` with a custom-patched `Linux 6.2.0` kernel for TDX support. The hypervisor is `QEMU 7.2.0`, also patched for TDX compatibility, and we use TDX tools from `tag 2023ww15` [23]. The Intel TME-MK functionality is enabled with auto bypass configured, allowing memory encryption only for TDs. We enable GPU passthrough to the VM (either traditional or TD) using the `Linux VFIO` driver [32]. Note that for TDs, the GPU must be set to CC mode (using NVIDIA GPU Admin Tools[5] [14]) to work properly. All CC experiments were conducted inside a TD, while non-CC experiments were performed in a regular VM—no experiments were run directly on the host. This methodology aligns with prior studies [16], [18].

[5]We set CC mode to `devtools` [13] to access performance counters.

To ensure stable performance measurements, hyper-threading and CPU auto-boost are disabled, and the CPU clock frequency is fixed at 2.1 GHz. NUMA balancing is turned off to avoid NUMA effects. Unless specified, the VM or TD is allocated 64GB of memory and pinned to NUMA node 0 (16 cores) using `numactl`, where the GPU is attached. Workload-specific settings and evaluation results are detailed in the following sections. Performance statistics are collected using NVIDIA Nsight Systems [87] and software timers.

V. GPU Performance Model

Figure 3 shows a high-level formulation of our performance model. Our performance model is inspired by the *Effective KLO* [88] work, which was originally targeted for mobile GPUs but did not account for data movement. As shown in Figure 3, we dissect total end-to-end application execution time (a metric to measure application performance, P) into four parts. The first part (A), termed as T_{mem}, primarily includes data transfer (e.g., H2D and D2H) overhead. After a launch operation completes, the GPU kernel is injected into a task queue [40], [89], where it waits for a time interval before execution begins; this is referred to as Kernel Queuing Time (KQT). Similarly, before the next consecutive launch can start, there is a waiting period named as Launch Queuing Time (LQT). The second part (B) of our performance model captures the combined duration of both kernel launch operations (KLO) and the time kernels spend waiting in the queue (LQT) before they are launched. The third part (C) considers the combined time related to kernel execution (KET) and associated queuing time (KQT). Finally, the last part (D), termed as T_{other}, primarily includes memory allocation (ALLOC), memory de-allocation (FREE), and synchronization overheads (SYNC). Regarding synchronization overhead, most of it may overlap with part (C), while the non-overlapping portions are included in this T_{other} category.

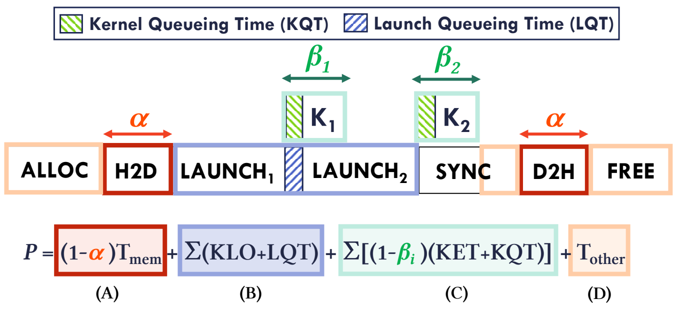

Kernel Queueing Time (KQT)
Launch Queueing Time (LQT)
β1
β2
α
α
K1
K2
ALLOC
H2D
LAUNCH1
LAUNCH2
SYNC
D2H
FREE
P = (1-α)Tmem + Σ(KLO+LQT) + Σ[(1-βi)(KET+KQT)] + Tother
(A)
(B)
(C)
(D)

Fig. 3: GPU Performance Model

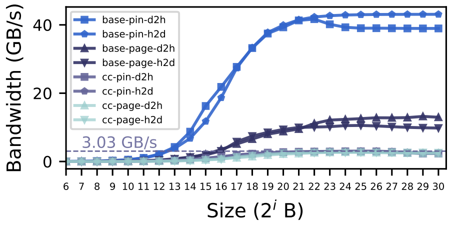
Note that α and β_i are parameters that indicate how much of the memory copy or kernel execution can overlap with other operations, respectively. For example, $\alpha = 0$ implies that the memory copy operations (e.g., H2D) are not overlapped with other operations. With the help of asynchronous APIs provided by CUDA (e.g., Streams), overlapping between different operations is possible (see Sec. VII-A), leading to a higher value of α. Regarding β_i, its value can be different for each kernel. For the scenario shown in Figure 3, β_1 for K1 is 1, implying that part (C) is completely overlapped with part (B). In contrast, β_2 for K2 is zero, implying that both parts (B and C) contribute toward total end-to-end application execution time (P). The maximum values of both α and β_i are 1.

VI. Performance Overheads of CC

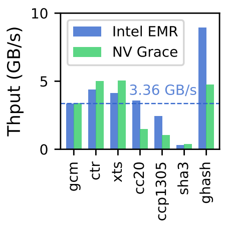
In this section, we evaluate the performance overheads of GPU-based confidential computing systems through a comprehensive set of benchmarks and analyses. We aim to understand how systems perform under CC, identify the sources of overhead, and provide a detailed breakdown of where these overheads arise. We will use the performance model discussed in Section V to drive our discussions.

A. Effect of CC on Data Transfer and Memory Management

Most GPU applications adopt a *copy-then-execute* scheme, which requires explicitly transferring the necessary data to GPU memory. It is because typically, CPU and GPU operate within isolated, non-coherent memory domains, and hence, such data movement is unavoidable. Moreover, because of the GPU's internal architecture, multiple memory copy operations cannot be multi-threaded (e.g., using CUDA APIs[6] or OpenMP) and thus become part of the critical path. As a result, data movement must be carefully considered when evaluating end-to-end performance.



(a) Data transfer bandwidth



(b) Crypto throughput

Fig. 4: (a) Data transfer bandwidth between the host (CPU) and the device (GPU). The X-axis is the transferred data size from 64 B to 1 GB. Two types of memory are investigated: pageable and pinned memory. Two computation modes are considered: *base*, which represents the non-CC mode (we interchangeably use the terms base, non-CC or CC-off), and *cc*, which refers to the confidential computing mode. (b) Single core throughput of encryption and authentication algorithms on different CPUs.

Data Transfer. We first evaluate the PCIe data transfer bandwidth, as shown in Fig. 4a. We observe that there is a significant bandwidth gap between CC and non-CC modes. Previous studies [18] have highlighted such gaps. However, they did not account for the type of memory used during the transfer. For pinned memory and pageable memory [90], the bandwidth gaps between CC and non-CC modes differ. In non-CC mode, pinned memory offers a bandwidth advantage as it reduces the need for additional memory copies. However, due to the memory isolation enforced by TDX [3], native pinned memory cannot be used in CC mode. As shown in the figure, the bandwidth for pinned and pageable memory is nearly identical in CC mode. This aligns with observations from previous work [3], which indicates that pinned memory in CC mode is implemented using pageable memory mechanisms through UVM.

Observation 1. *The PCIe bandwidth utilization in CC mode drops significantly compared to non-CC mode. Additionally, the bandwidth gap between pageable and pinned memory observed in non-CC mode disappears in CC mode, suggesting that pinned memory relies on pageable mechanisms in CC mode.*

To understand the PCIe data transfer bandwidth gap between CC and non-CC modes, we study the encryption process and its associated bandwidth. Current CC implementations leverage `AES-NI` acceleration. Interestingly, as shown in Fig. 4b, we find that the maximum data transfer bandwidth (3.03 GB/s, pin-h2d) under CC is slightly lower than the maximum AES-GCM throughput (3.36 GB/s). The reason behind this gap may be explained by analyzing the data transfer process involved. In CC systems, data transfers require a *bounce buffer*, which is a shared memory region managed by the hypervisor. Consequently, the corresponding extended page table (EPT) is also managed by the hypervisor. The copy process can be described in the following steps: a) prepare the data in the TDX-isolated memory region, whose EPT is managed by the TDX Module, b) encrypt the data using software AES-GCM, c) copy the encrypted data to the bounce buffer, d) GPU's DMA engines copy the data from the bounce buffer to GPU, and e) data is then decrypted and stored in GPU HBM. In summary, these context switches and data movement could be contributing to the bandwidth gap.

We also evaluate the performance of AES-GCM and other crypto algorithms on two CPUs: an Intel EMR CPU and an NVIDIA Grace CPU. Our results are shown in Fig. 4b. Unfortunately, the encryption performance of AES-GCM on both CPUs remains much lower than the peak bandwidth obtained in non-CC scenarios. GHASH, which could be used to construct GMAC [44], achieves higher throughput (up to 8.9 GB/s) at the cost of confidentiality. As suggested by prior work [18], TEE-IO [91] technology offers a potential solution to this problem. However, its adoption requires hardware replacement, which may incur high costs. Therefore, software-level optimization techniques must be explored and applied. We discuss some existing optimizations in Sec. VII.

Observation 2. *The absence of dedicated hardware AES engines results in low encryption throughput. Even with AES-NI acceleration, encryption throughput remains insufficient to meet performance demands. While alternative cryptographic algorithms may offer higher throughput, they often come at the cost of weaker security guarantees.*

[6]CUDA memory copy APIs are blocking in nature.

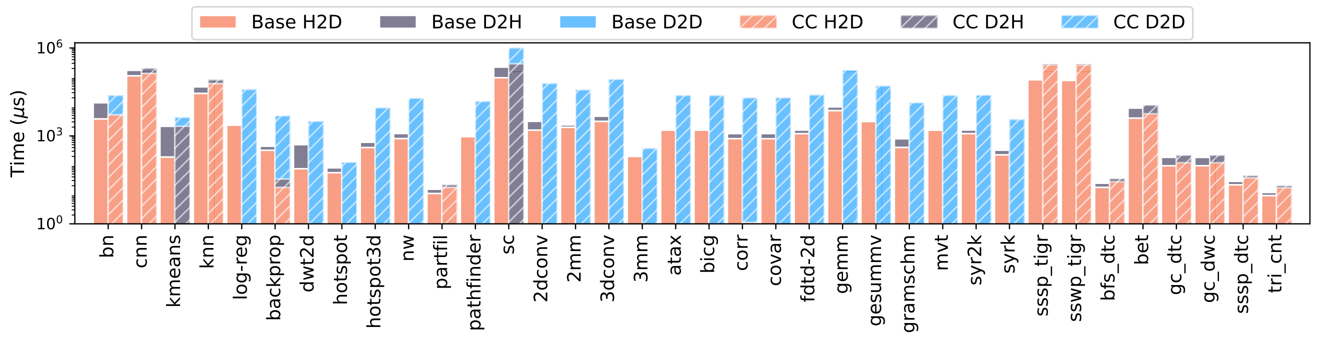


Fig. 5: Time spent on copy operations in Base and CC modes. The hatched bars represent CC mode. H2D, D2H, and D2D correspond to host-to-device, device-to-host, and device-to-device transfers, respectively. The Y-axis is on a logarithmic scale.

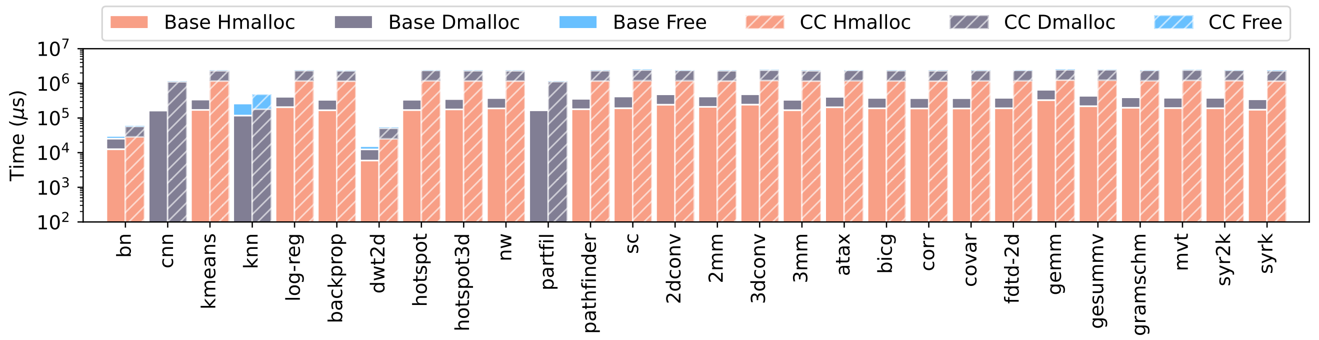


Fig. 6: Time spent on memory allocation and deallocation. The hatched bars represent CC mode. Hmalloc, Dmalloc, and Free correspond to `cudaMallocHost`, `cudaMalloc`, and `cudaFree`, respectively. The Y-axis is on a logarithmic scale.

To better understand how data transfer impacts application-level performance, we analyze detailed memory copy events across several benchmark suites. Specifically, we select applications from Rodinia [92], Polybench [93], and UVM-Bench [55], as well as two graph processing suites, Graph-BIG [94] and Tigr [95]. Fig. 5 shows the time spent on memory copy operations. It is important to note that in some applications, the underlying copy mechanism may change under CC. For example, D2D transfers may occur in certain CC applications (e.g., `2dconv`) but not in the non-CC version. In these cases, H2D or D2H copies on pinned memory are identified as D2D operations by Nsight Systems, which also labels them as *Managed*. It is likely because these copy operations are performed on pinned memory which are achieved via page faults and encrypted data migrations (**Observation 1**). We refer such UVM mechanism under CC as encrypted paging. Across all evaluated applications, copy time increases significantly under CC, with overheads ranging from 1.17× in `cnn` to 19.69× in `2dconv`.

Observation 3. *On average, copy operations in CC mode take 5.80× longer compared to non-CC mode, with a maximum slowdown of 19.69×. Pinned memory is converted to UVM encrypted paging in CC mode, which incurs high overhead.*

In summary, data transfer affects overall performance (Figure 3). Under CC, T_{mem}, is further increased due to software-based encryption and the UVM mechanism. To optimize T_{mem}, two approaches can be considered: a) overlapping memory copy operations with computation (i.e., by increasing α), and b) increasing bandwidth and/or reducing data movement between CPU-GPU (i.e., by reducing T_{mem}).

Memory (De)allocation. Although memory management (i.e., allocation and deallocation) is part of T_{other}, it significantly impacts end-to-end performance. Since TDX isolates memory management functionalities, these operations take longer in CC mode. As shown in Fig. 6, both allocation and deallocation incur higher overhead due to isolation stacks. On average, in CC mode, `cudaMalloc`, `cudaMallocHost`, and `cudaFree` take 5.67×, 5.72×, and 10.54× longer, respectively, compared to the non-CC setup. Since in UVM settings, `cudaMallocManaged` is used to manage the unified memory space, we measured its performance under CC as well. The results show that `cudaMallocManaged` under CC incurs a 5.43× slowdown compared to the non-CC version. Meanwhile, free managed memory with `cudaFree` experiences a 3.35× slowdown. We further compared the time spent on memory allocation and deallocation between non-UVM and UVM applications. Using the non-CC, non-UVM setup as the baseline, we observe that non-CC UVM reduces allocation time to 0.51× but increases deallocation time to 3.13×; meanwhile, CC-UVM takes 1.01× longer for allocation and 18.20× longer for deallocation. Hence, optimizing memory management under TDX presents a potential opportunity.

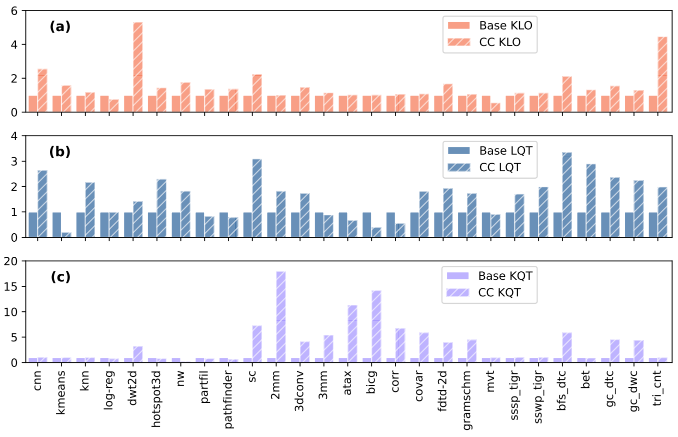


Fig. 7: Effect of CC on Kernel Launch Overhead (KLO), Launch Queuing Time (LQT), and Kernel Queuing Time (KQT). Applications with no queuing time (e.g., only a single launch) are excluded. Results are normalized to non-CC time.

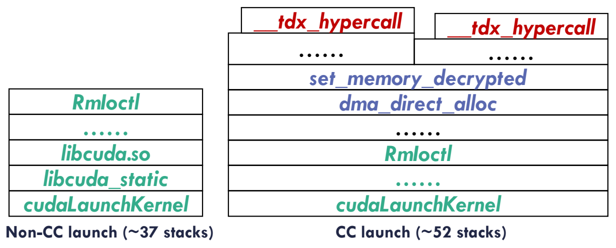


Fig. 8: Simplified call stack derived from a flame graph [96] of a `cudaLaunchKernel` call inside a TD, obtained using *perf* [97]

B. *Effect of CC on Kernel Launch and Execution*

Fig. 7 shows the effect of CC on Kernel Launch Overhead (KLO), Launch Queuing Time (LQT) and Kernel Queuing Time (KQT). Our observations are described below:

Kernel Launch Overhead (KLO). In some applications (e.g., `dwt2d`), KLO significantly delays the start of a kernel execution, and this overhead further increases in CC mode. For instance, as shown in Fig. 7a, KLO increases by up to 5.31× in `dwt2d`, which only involves 10 kernel launches. To further confirm the general behaviors of KLO under CC, we show the Cumulative Density Functions (CDFs) of each launch operation in Fig. 11a. We observe the distribution of CC KLO shifts to the right compared to Base, and the average CC KLO is higher than Base KLO. To further understand this increase, we use *perf* to profile a single kernel launch and generate its call stacks from a flame graph, a simplified version is shown in Fig. 8. TDX introduces additional overhead when the TD interacts with external components (e.g., the TDX module and hypervisor) through hypercalls. This overhead also affects kernel launch operations, as the GPU driver must communicate with the device to configure execution. Such communication is mediated by the host via hypercalls to facilitate context switching. A comparison of the call stacks reveals a significant increase in TDX-related operations in CC mode. For example, the `dma_direct_alloc` call attempts to allocate a bounce buffer, while `set_memory_decrypted` converts private memory into shared memory. According to hypercall evaluations [16], `tdx_hypercall` operations can increase latency by over 470%, further exacerbating the observed overhead.

Queuing Time: LQT and KQT. As shown in Fig. 7b, for applications like `sc` and `3dconv`, the LQT increases substantially over a non-CC setup. For example, `3dconv` involves 254 launches of the same kernel within a loop. There are 1611 launches for `sc`. The accumulated queuing time between consecutive kernel launches becomes significant and worsens under CC. Interestingly, some applications exhibit lower LQT in CC mode. Applications such as `3mm`, `atax`, `bicg`, and `corr` involve only 2–4 short-running kernel launches, where potential queuing time variations are not stable and can fluctuate. We believe `kmeans` is an outlier since its first non-CC launch is significantly ahead of its second launch. Some applications exhibit significantly increased KQT under CC, such as `2mm` and `sc` ,which are shown in Fig. 7c. For `2mm`, which only has 2 kernel launches, the KQT is minimal in non-CC mode, making it highly susceptible to amplification under CC. This behavior also applies to applications with fewer kernel launches, such as `3mm`, `atax`, `bicg`, and `corr`, all of which show noticeably increased KQT under CC.

Observation 4. *On average, CC increases KLO by 1.42× mostly due to TDX hypercalls. However, its impact on LQT and KQT largely depends on the number of kernel launches. For applications with a low number of kernel launches, KQT can be significantly amplified. On average, CC increases LQT by 1.43× and KQT by 2.32×.*

Kernel Execution Time (KET). If there is no further communication with the host, the KET should have minimal performance degradation under CC. Fig. 9 shows KET results across several benchmarks. We observe that CC non-UVM KETs are nearly identical compared to non-CC non-UVM KETs across all evaluated applications. We further plot the CDFs of KETs, as shown in Fig. 11b. The results indicate that KET follows (almost) the same distribution even under CC. When UVM is applied, even in the non-CC setting, it results in an average slowdown of 5.29×. Due to the UVM encrypted paging overhead under CC, KET gets amplified: it increased from 1.08× (`gramschm`) to 164030.65× (`2dconv`).

Observation 5. *CC has minimal impact on non-UVM kernels. The average KET across the evaluated applications shows only a 0.48% increase in CC mode. However, encrypted page migration of UVM in CC mode incurs an average slowdown of 188.87×.*

Case Study. We demonstrate how KLO and LQT impact non-UVM application performance under CC. We exclude the impact of KQT in this analysis as most KQT durations

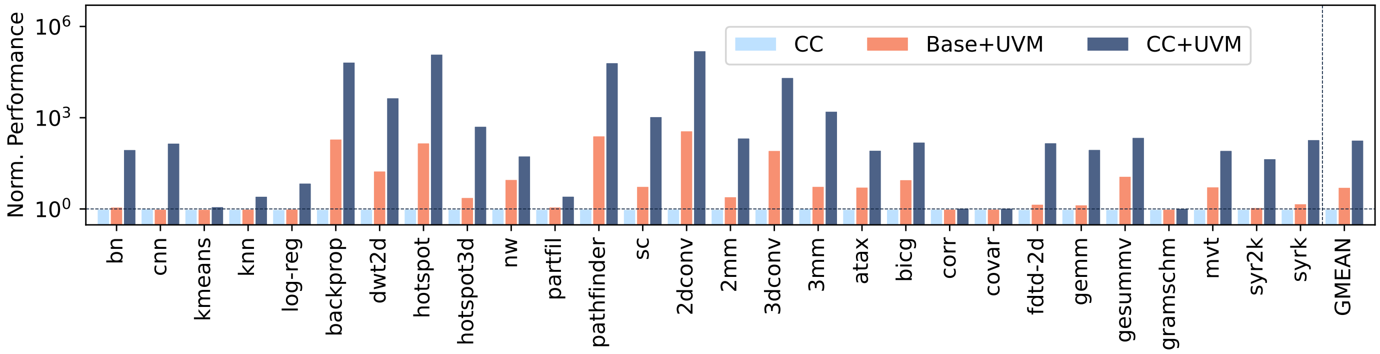


Fig. 9: KET on the GPU. The results are normalized to the non-CC, non-UVM baseline. The Y-axis is on a logarithmic scale.

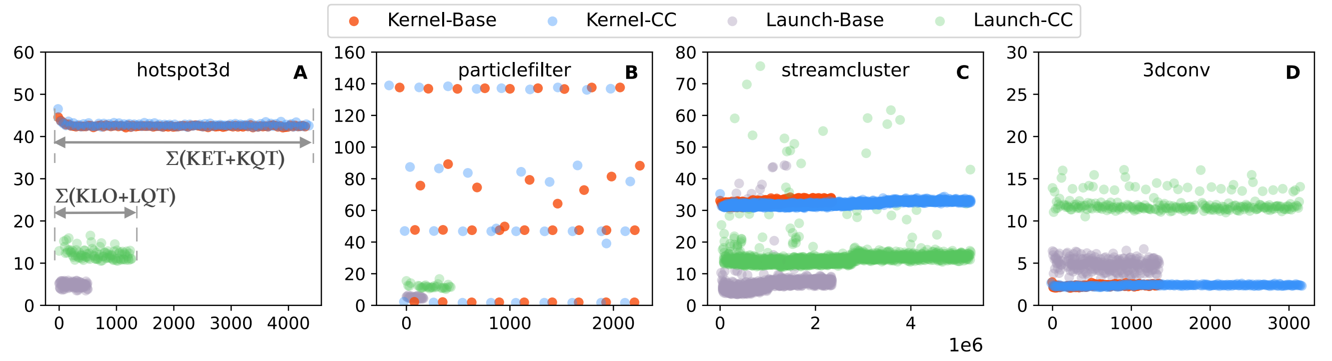


Fig. 10: Distribution of events and their durations for representative applications. We focus on two types of events: Kernel and Launch. The X-axis represents event start timestamps (µs). The Y-axis represents the event duration (µs): it can be either KET (for a Kernel event) or KLO (for a Launch event). The events with the longest duration are excluded for clarity.

are in the order of tens of microseconds, which is relatively small compared to the total execution timeline (often spanning millions of microseconds). Fig. 10 presents the kernel launch and kernel execution events across the application lifetime. We include four applications in this analysis. Memory copy operations are not considered in this context as all these applications currently have $\alpha = 0$ (no overlapping).

The trace in Fig. 10A suggests that although KLO and LQT increase (with **CC launch** points shifted upward on the Y-axis and sparsely distributed along the X-axis compared to **non-CC launch**), the overall execution time of this application does not increase significantly. This is because $\sum(\mathrm{KLO}+\mathrm{LQT})$ can be overlapped by the long KET. A similar situation is observed in Fig. 10B, where the launch events under CC exhibit higher KLO and more sparse distributions. However, the diverse execution times of a large number of kernels (in **orange** and **blue** for non-CC and CC, respectively) effectively hide the impact of KLO and LQT.

For the applications above, we further explain their behaviors based on Fig 3. When the term $\sum(\mathrm{KLO}+\mathrm{LQT})$ increases, the value of β also increases. Since $\sum$(KET+KQT) is sufficiently large, it helps balance the overall time. As a result, the end-to-end performance is less affected. On the other hand, for applications like `streamcluster` (sc) (Fig. 10C) and `3dconv` (Fig. 10D), the Kernel-to-Launch Ratio (KLR) is low. KLR is defined as $\mathrm{KET}/(\mathrm{KLO}+\mathrm{LQT})$. Low KLR will lead to β closer to 1, which means kernel execution is largely hidden by launch. In these cases, performance is dominated by KLO and LQT. Note that a sufficient number of kernels is needed to observe this behavior.

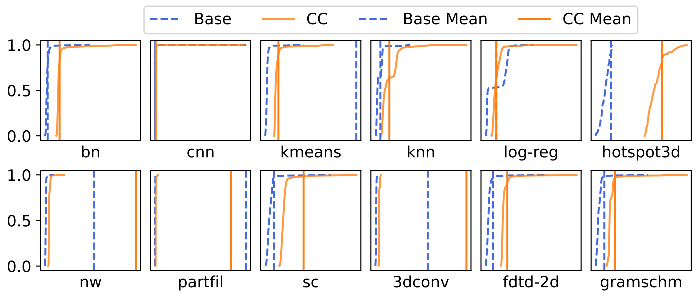
Observation 6. *For applications with many kernel launches, a high Kernel-to-Launch Ratio (KLR) helps. A high KLR means $KET/(KLO+LQT)$ is large. In this case, under CC, the impact from launch is small because launch events could be overlapped by other kernel events. But for applications with low KLR, launch events become sparse and β will approach to 1. This leads to longer finish times and term $\sum(KLO+LQT)$ will then dominate the performance.*

VII. Techniques to Address Overheads of CC

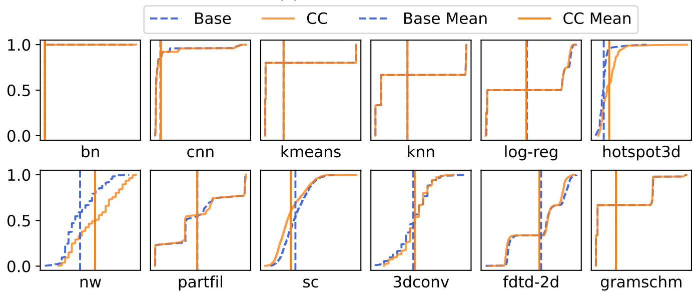
In this section, we explore optimization opportunities to address CC overheads. While some optimizations are based on existing techniques, this is the first time they are analyzed in the context of CC systems, providing novel perspectives on addressing CC-specific challenges.

A. Kernel Fusion and Overlapping

We consider kernel fusion and overlapping mechanisms to reduce and hide the overheads of CC. To explore these optimizations, we start by studying a microbenchmark shown in Listing. 1. This microbenchmark employs PTX [98] `nanosleep` instructions to create a kernel with a fixed



(a) KLO CDFs



(b) KET CDFs

Fig. 11: CDFs for KLO and KET. The X-axis represents the event duration (in µs). For clarity, exact values are not shown. Note that for the kernel launch CDF, some launches span different orders of magnitude. To display the CDF on the same scale, the top 5 longest launch durations are removed. The average value is calculated over all data points, without any removals.

```
__global__ void K_x(){
    #pragma unroll N_x
    for (int i = 0; i < ITER_x; i++) {
        asm volatile("nanosleep.u32 %0;"::
        ↪   "r"(CLK));
    }
}
```

Listing 1: Code for a microbenchmark kernel that runs for a specified duration using PTX `nanosleep` instructions. We instantiate multiple kernels with different values of x to control KET.

```
int main(){
    // ......
    for (int i = 0; i < nStreams; ++i){
        cudaMemcpyAsync(&d_a[offset], &a[offset],
        ↪   bytes, H2D, s[i]);
    }
    for (int i = 0; i < nStreams; ++i){
        K_x<<<TH, BLK, 0, s[i]>>>(d_a, offset);
    }
    for (int i = 0; i < nStreams; ++i){
        cudaMemcpyAsync(&a[offset], &d_a[offset],
        ↪   bytes, D2H, s[i]);
    }
}
```

Listing 2: Host code attempts to overlap data transfer with kernel computation. This host code launches kernels based on the number of streams.

execution time (100 ms in our setup) [99]. To control the size of the generated machine code (PTX and SASS), we utilized a loop unrolling parameter, $\mathtt{N_x}$.

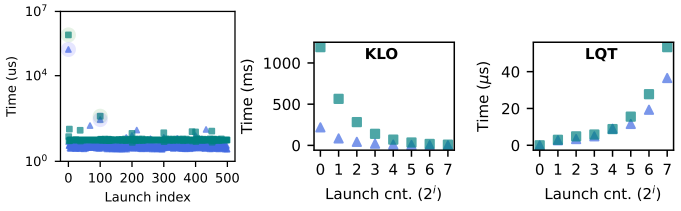
Kernel Fusion. We launched two microbenchmark kernels (shown in Listing 1), K_0 and K_1, back-to-back: K_0 is launched 100 times, then followed by K_1 for 100 times. KLO is shown in Fig. 12a. It is evident that the first launch of a new kernel incurs a higher overhead. Subsequent launches present lower KLO. As shown in Fig. 10, the number of kernel launches in some applications introduces significant overhead (**Observation 6.**). A straightforward optimization is to fuse some kernels, reducing the number of launches and increasing the execution time of individual kernels. This can lower the term KLO + LQT, shortening runtime and increasing the Kernel-to-Launch Ratio (KLR). While kernel/launch fusion has been extensively studied in other contexts [100]–[105], its benefits and trade-offs for CC remain unexplored.

Unfortunately, fusing all kernels into a single one is not ideal, as fewer launches can result in significantly higher KLO as shown in Fig. 12a. To evaluate this, we keep the total KET the same and progressively fuse kernels down to a single launch. The results, shown in Fig. 12b, reveal that KLO and LQT follow different trends during fusion, highlighting a trade-off between the number of kernels fused and CC performance. It also indicates that a fully fused kernel is suboptimal. A balanced approach to kernel fusion is needed for optimal performance under CC.

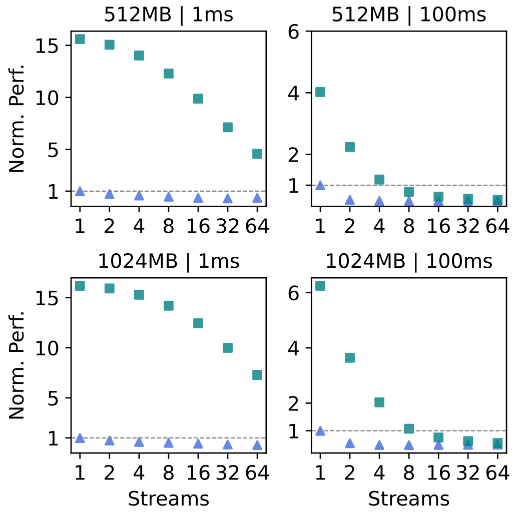
Kernel fusion typically requires source code modification. For applications like `3dconv`, where a single kernel is executed iteratively, a more efficient approach is to fuse launch operations using `cudaGraph` APIs [106]. However, there is a trade-off between graph creation and KLO, making the optimal fusion level a key consideration. Ekelund et al. [107] found that an optimal fusion point exists for kernel launches, independent of the application. However, whether this finding holds in CC mode remains unclear, and we leave it for future work.

Observation 7. *Launch count affects KLO, the first few launches show much higher KLO value. Due to the different trends of KLO and LQ when the number of launches change, kernel fusion under CC is not a trivial task, and it has different objectives compared to non-CC world.*

Overlapping. Beyond fusion, another potential optimization is to improve overlapping between operations (increasing α and β). While overlapping [108], [109] can be difficult due to data dependencies, we study an ideal case where kernels have no data requirements (Listing. 2). Results in Fig. 12c show data transfer sizes of 512MB and 1GB accumulated across all 64 streams with KET of 1ms and 100ms. We observe that overlapping is harder to achieve when: a) CC is used, and b) KET is short. In both cases, increasing KET (and hence Compute-to-IO ratio) improves overlap. Adding streams also helps, however, its benefits are limited under CC.



(a) KLO and launch id.   (b) Implication of launch count.



(c) Overlapping and performance.

Fig. 12: Microbenchmark results considering number of launches, kernel size, and overlapping. Across all figures, triangle (△) stands for Base and square (□) stands for CC.

Observation 8. *Overlapping can improve CC performance by hiding data movement operations (i.e., encryption overheads). Ideally, increasing the KET to achieve a higher compute-to-IO ratio may enhance overlap efficiency.*

B. Quantization

After the characterization and analysis of various GPGPU benchmarks and microbenchmarks, in this section, we focus on characterizing convolutional neural networks (CNNs) and large language models (LLMs). These workloads may process private data, but in this study, we are using open-source data, and training is done on public models.

CNN Workloads. We evaluate six CNN models—VGG16 [110], ResNet50 [111], MobileNetv2 [112], SqueezeNet [113], Attention92 [114], and Inception-v4 [115]—trained on the CIFAR-100 [116] dataset for 200 epochs. Using throughput (images per second) and training time as metrics, we analyze CC performance compared to the non-CC FP32 training baseline (Fig. 13).

With a batch size of 64 and CC on, throughput drops up to 36% (average 24%), and training time increases up to 53% (average 31%). Increasing the batch size to 1024 significantly reduces overhead, with an average loss in throughput of 7.3% and an increase in training time by 6.7%. We further

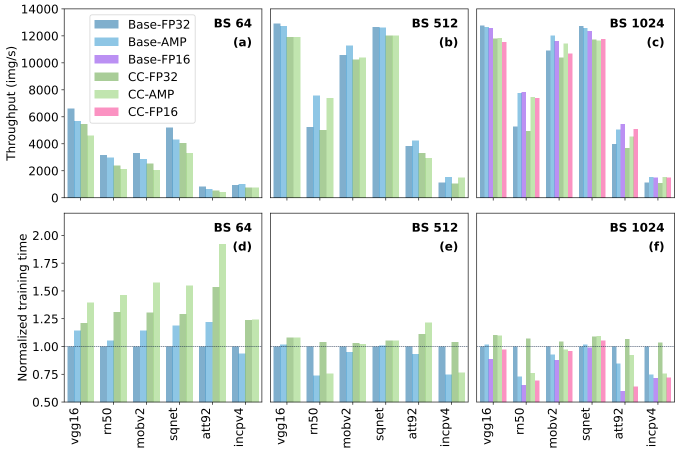


Fig. 13: CNN training throughput and training time for different batch sizes under CC and non-CC modes. For a batch size of 1024, `FP16` quantized training is applied. Training time is normalized to the non-CC `FP32` training time.

investigated two quantization optimizations. Automatic Mixed Precision (AMP) [117], [118], which is commonly used during training, worsens performance for small batch sizes due to additional computations (e.g., precision casting). With a batch size of 64, AMP reduces CC throughput by up to 50% (average 19.7%) and increases training time by up to 92% (average 50.9%). However, AMP becomes effective with larger batch sizes. For example, with a batch size of 1024, AMP outperforms the non-CC AMP baseline, increasing throughput by up to 40.8% (average 11.8%) and reducing training time by up to 24.4% (average 7.8%). Since AMP does not significantly reduce the amount of data transferred between the CPU and GPU, we applied `FP16` quantization [119] to the largest batch size. This further reduced training time by up to 46.1% (average 27.7%).

LLM Workloads. To demonstrate the impact of CC on LLMs, we evaluate the inference throughput under different configurations. Specifically, we deploy the Meta-Llama-3-8B [120] model and tested two inference backends: HuggingFace (HF) [121] and vLLM [122]. Since the model uses 16-bit parameters, we further evaluated a 4-bit quantization solution, Activation-aware Weight Quantization (AWQ) [123], which selectively quantizes the weights.

Fig. 14 shows the throughput speedup of vLLM compared to `BF16|CC-off|HF` baseline at the same configuration. Throughput is measured as the number of tokens generated per second for batched requests. All data points represent the median of three runs. It shows that vLLM consistently outperforms (i.e., all numbers shown in Fig. 14 are larger than one) HF in throughput across all configurations, even when CC is enabled. In general, we make two additional observations. First, CC-on is worse than CC-off for both `BF16` and `AWQ`. Second, throughput of `AWQ` is higher than `BF16`. Interestingly, `BF16` performs better than `AWQ` for larger batch sizes (64 and 128) [124]. In some cases, such as with batch size 8, the `BF16` model even performs better with CC-on than with CC-off.

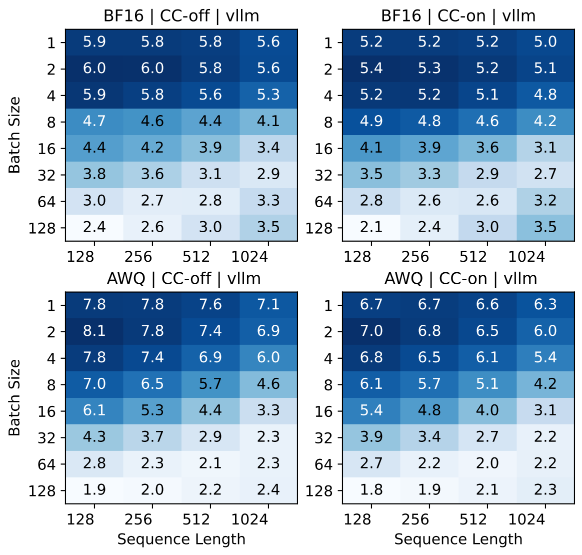


Fig. 14: Throughput speedup of the vLLM serving framework for the Llama-3-8B model. All values are compared to the HuggingFace non-quantized CC-off baseline. The BF16 model represents the non-quantized configuration, while AWQ indicates the 4-bit Activation-aware Weight Quantization.

Observation 9. *For CNNs, quantization (i.e., `FP16`) reduces training time as it cuts down the amount of data transferred between CPU and GPU. As for LLMs, the serving backend significantly affects inference throughput. Overall, vLLM outperforms HF in both quantized and non-quantized models and remains robust with CC enabled.*

VIII. Related Work and Discussions

To the best of our knowledge, this is the first work that dissects the performance overheads of GPU-based confidential computing and evaluates optimizations to address those overheads. In this section, we discuss some additional prior works related to this paper.

Optimizing Software Encryption. Encryption of CPU-GPU data transfers is implemented using OpenSSL. Although it leverages `AES-NI` for hardware-accelerated encryption, it remains a single-threaded process, making encryption inefficient. However, directly enabling multi-threading at the CUDA API level is not feasible, as such calls would block the application. Tan et al. [19], [125] modified the OpenSSL and CUDA memory copy API implementations at the runtime library level, allowing them to use multiple worker threads to perform encryption in parallel. They further overlap the (de)encryption process with GPU execution to reduce overhead. Similar approaches have been explored by Wang et al. [126]. They proposed several optimizations to increase the parallelism of authentication and overlap GPU cryptographic kernels with PCIe transmission. GPU cryptographic kernels have also been used in HIX [11] and LITE [127] to construct customized GPU TEEs. However, since these techniques are implemented outside the NVIDIA H100 CC, it remains unclear how they can be integrated into the CC framework.

TEE and Device. Many studies have explored customized TEEs for external devices. For example, IceClave [128] introduced a TEE for SSDs to enable secure in-storage computing, utilizing a Bonsai Merkle Tree (BMT) with a hybrid-counter scheme for memory encryption and verification. Other SSD encryption schemes, such as D-Shield [129], have also been proposed. Since SSDs are a potential solution to the GPU memory wall problem [130], integrating them into the CC framework could benefit large memory footprint applications. Similarly, efforts have been made to secure Network Interface Cards (NICs). Zhou et al. proposed S-NIC [131], which isolates network functions for secure outsourcing. Li et al. introduced Bifrost [17], integrating the Mellanox Connect-X6 NIC into AMD CVM and optimizing the CVM-IO tax with minimum modifications to the kernel. A major contributor to this tax, as discussed in the TDX-based GPU CC context, is the bounce buffer. Additionally, recent work has explored scaling counter-mode encryption for multi-GPU networks [132].

IX. Conclusions

In this paper, we characterize and analyze the performance of GPU-based confidential computing (CC) systems, uncovering substantial overheads in data transfer, memory management, encryption, and kernel launches. Through detailed analysis, we investigate optimization techniques—including kernel fusion, overlapping, and quantization—to reduce CC-induced overheads, while discussing the trade-offs involved. We hope our findings and insights contribute to a deeper understanding of CC on GPU-based systems and guide future efforts in performance optimization.

Code and Data Availability

The code, scripts, and data associated with this paper can be found here: https://github.com/insight-cal-uva/hcc-ispass25-artifact. This paper is solely focused on performance evaluation and does not uncover any new security vulnerabilities. All data, models, and tools used in this paper are based on publicly available projects.

Acknowledgments

We thank the anonymous reviewers of ISPASS 2025 for their detailed and constructive feedback. We also thank the members of the Insight Computer Architecture Lab at the University of Virginia (UVA) for their helpful comments. This work was supported by a start-up grant from UVA and was performed using computing facilities at UVA.

References

[1] Confidential computing consortium. https://confidentialcomputing.io/. Accessed: 2024-12-09.

[2] Dominic P Mulligan, Gustavo Petri, Nick Spinale, Gareth Stockwell, and Hugo JM Vincent. Confidential computing—a brave new world. In *2021 international symposium on secure and private execution environment design (SEED)*, pages 132–138. IEEE, 2021.

[3] Gobikrishna Dhanuskodi, Sudeshna Guha, Vidhya Krishnan, Aruna Manjunatha, Michael O'Connor, Rob Nertney, and Phil Rogers. Creating the first confidential gpus: The team at nvidia brings confidentiality and integrity to user code and data for accelerated computing. *Queue*, 21(4):68–93, 2023.

[4] Intel Corporation. Intel® software guard extensions (intel® sgx). https://community.intel.com/legacyfs/online/drupal_files/332680-002.pdf, June 2015. Reference Number: 332680-002, Revision Number: 1.1, Accessed: 2024-12-09.

[5] Intel Corporation. Intel trust domain extensions (intel tdx) overview. https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html. Accessed: 2024-12-09.

[6] Advanced Micro Devices (AMD). Amd sev-snp: Strengthening vm isolation with integrity protection and more. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, 2020. White Paper.

[7] Arm Limited. Arm confidential compute architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture. Accessed: 2024-12-09.

[8] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[9] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the PENGLAI enclave. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 275–294. USENIX Association, July 2021.

[10] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on gpus. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 681–696, 2018.

[11] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous isolated execution for commodity gpus. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 455–468, 2019.

[12] NVIDIA Corporation. Nvidia confidential computing. https://www.nvidia.com/en-us/data-center/solutions/confidential-computing/. Accessed: 2024-12-09.

[13] NVIDIA Corporation. Intel tdx - confidential computing deployment guide. https://docs.nvidia.com/cc-deployment-guide-tdx.pdf. Accessed: 2024-12-09.

[14] NVIDIA Corporation. Nvidia gpu admin tools. https://github.com/NVIDIA/gpu-admin-tools/tree/main. Accessed: 2024-12-09.

[15] Victor Costan. Intel sgx explained. *IACR Cryptol, EPrint Arch*, 2016.

[16] Misanori Misono, Dimitrios Stavrakakis, Nuno Santos, and Pramod Bhatotia. Confidential vms explained: An empirical analysis of amd sev-snp and intel tdx. In *Proceedings of the 2025 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '25)*, 2025.

[17] Dingji Li, Zeyu Mi, Chenhui Ji, Yifan Tan, Binyu Zang, Haibing Guan, and Haibo Chen. Bifrost: Analysis and optimization of network {I/O} tax in confidential virtual machines. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 1–15, 2023.

[18] Apoorve Mohan, Mengmei Ye, Hubertus Franke, Mudhakar Srivatsa, Zhuoran Liu, and Nelson Mimura Gonzalez. Securing ai inference in the cloud: Is cpu-gpu confidential computing ready? In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, pages 164–175. IEEE, 2024.

[19] Yifan Tan, Cheng Tan, Zeyu Mi, and Haibo Chen. Pipellm: Fast and confidential large language model services with speculative pipelined encryption. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2025.

[20] Kapil Vaswani, Stavros Volos, Cédric Fournet, Antonio Nino Diaz, Ken Gordon, Balaji Vembu, Sam Webster, David Chisnall, Saurabh Kulkarni, Graham Cunningham, et al. Confidential computing within an ai accelerator. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 501–518, 2023.

[21] Intel Corporation. Tdx support for intel xeon processors. https://www.intel.com/content/www/us/en/support/articles/000091103/processors/intel-xeon-processors.html. Accessed: 2024-12-09.

[22] Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. Intel trust domain extensions (tdx) security review. Technical report, Google technical report, 2023.

[23] Intel Corporation. Intel tdx tools. https://github.com/intel/tdx-tools.

[24] Shay Gueron. Memory encryption for general-purpose processors. *IEEE Security & Privacy*, 14(6):54–62, 2016.

[25] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel tdx demystified: A top-down approach. *ACM Computing Surveys*, 56(9):1–33, 2024.

[26] Intel Corporation. Runtime encryption of memory with intel® total memory encryption–multi-key (intel® tme-mk). https://www.intel.com/content/www/us/en/developer/articles/news/runtime-encryption-of-memory-with-intel-tme-mk.html. Accessed: 2024-12-09.

[27] Intel® architecture memory encryption technologies specification. Technical Report Revision 1.5, Intel Corporation, October 2024. Accessed: 2024-12-09.

[28] National Institute of Standards and Technology. Recommendation for block cipher modes of operation: The xts-aes mode for confidentiality on storage devices. NIST Special Publication 800-38E, U.S. Department of Commerce, National Institute of Standards and Technology, 2010. Accessed: 2024-12-09.

[29] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. Vault: Reducing paging overheads in sgx with efficient integrity verification structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 665–678, 2018.

[30] Joshua Bakita and James H Anderson. Demystifying nvidia gpu internals to enable reliable gpu management. In *Proceedings of the 30th IEEE Real-Time and Embedded Technology and Appli cations Symposium*. IEEE, 2024.

[31] Yuanqing Miao, Yingtian Zhang, Dinghao Wu, Danfeng Zhang, Gang Tan, Rui Zhang, and Mahmut Taylan Kandemir. Veiled pathways: Investigating covert and side channels within gpu uncore. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1169–1183. IEEE, 2024.

[32] The Linux Kernel Documentation Project. Vfio - virtual function i/o documentation. https://www.kernel.org/doc/html/v5.9/driver-api/vfio.html. Accessed: 2024-12-09.

[33] Intel Corporation. Intel tdx module 1.5 base specification. Technical report, Intel Corporation, 2025. Accessed: 2025-02-27.

[34] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev:first-class gpu resource management in the operating system. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 401–412, 2012.

[35] Shinpei Kato. Implementing open-source cuda runtime. In *Proc. of the 54the Programming Symposium*, 2013.

[36] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. Gpuvm: Why not virtualizing gpus at the hypervisor? In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 109–120, 2014.

[37] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. Gpuvm: Gpu virtualization at the hypervisor. *IEEE Transactions on Computers*, 65(9):2752–2766, 2015.

[38] Hong-Cyuan Hsu and Che-Rung Lee. G-kvm: a full gpu virtualization on kvm. In *2016 IEEE International Conference on Computer and Information Technology (CIT)*, pages 545–552. IEEE, 2016.

[39] Xiaolong Wu, Dave Jing Tian, and Chung Hwan Kim. Building gpu tees using cpu secure enclaves with gevisor. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 249–264, New York, NY, USA, 2023. Association for Computing Machinery.

[40] Nathan Otterness and James H Anderson. Exploring amd gpu scheduling details by experimenting with "worst practices". In *Proceedings of the 29th International Conference on Real-Time Networks and Systems*, pages 24–34, 2021.

[41] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. Tunnels for bootlegging: Fully reverse-engineering gpu tlbs for challenging isolation guarantees of nvidia mig. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 960–974, New York, NY, USA, 2023. Association for Computing Machinery.

[42] Joshua Bakita and James H Anderson. Hardware compute partitioning on nvidia gpus. In *2023 IEEE 29th Real-Time and Embedded*

Technology and Applications Symposium (RTAS), pages 54–66. IEEE, 2023.

[43] Rob Nertney. The developer's view to secure an application and data on nvidia h100 with confidential computing. https://www.nvidia.com/en-us/on-demand/session/gtcspring23-s51684/. GTC Spring 2023, Accessed: 2024-12-09.

[44] Morris Dworkin. Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac. NIST Special Publication 800-38D, National Institute of Standards and Technology (NIST), November 2007. Accessed: 2024-12-09.

[45] OpenSSL Software Foundation. Openssl. https://github.com/openssl/openssl. Accessed: 2024-12-09.

[46] Intel Corporation. Intel® advanced encryption standard instructions (aes-ni). https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html. Accessed: 2024-12-09.

[47] Erhu Feng, Dahu Feng, Dong Du, Yubin Xia, Wenbin Zheng, Siqi Zhao, and Haibo Chen. siopmp: Scalable and efficient i/o protection for tees. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1061–1076, 2024.

[48] David S. Miller, Richard Henderson, and Jakub Jelinek. *Dynamic DMA Mapping Guide*. The Linux Kernel Documentation, 2025. Accessed: 2025-02-27.

[49] James E. J. Bottomley. *Dynamic DMA Mapping using the Generic Device*. The Linux Kernel Documentation, 2025. Accessed: 2025-02-27.

[50] NVIDIA Corporation. *CUDA C Programming Guide*. NVIDIA. Accessed: 2024-12-09.

[51] NVIDIA Corporation. Nvidia linux open gpu kernel module source. https://github.com/NVIDIA/open-gpu-kernel-modules/tree/main/kernel-open/nvidia-uvm. Accessed: 2024-12-09.

[52] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W Keckler. Towards high performance paged memory for gpus. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 345–357. IEEE, 2016.

[53] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. Batch-aware unified memory management in gpus for irregular workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1357–1370, 2020.

[54] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. Interplay between hardware prefetcher and page eviction policy in cpu-gpu unified virtual memory. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 224–235, 2019.

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

[56] Tyler Allen and Rong Ge. In-depth analyses of unified virtual memory system for gpu accelerated computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[57] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting unprotected I/O operations in AMD's secure encrypted virtualization. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1257–1272, Santa Clara, CA, August 2019. USENIX Association.

[58] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656. IEEE, 2015.

[59] Yuankun Zhu, Yueqiang Cheng, Husheng Zhou, and Yantao Lu. Hermes attack: Steal dnn models with lossless inference accuracy. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[60] Xing Hu, Ling Liang, Lei Deng, Shuangchen Li, Xinfeng Xie, Yu Ji, Yufei Ding, Chang Liu, Timothy Sherwood, and Yuan Xie. Neural network model extraction attacks in edge devices by hearing architectural hints. *arXiv preprint arXiv:1903.03916*, 2019.

[61] PCI-SIG. Integrity and data encryption (ide) ecn deep dive. https://pcisig.com/sites/default/files/files/PCIe%20Security%20Webinar_Aug%202020_PDF.pdf. Accessed: 2024-12-09.

[62] PCI-SIG. Integrity and data encryption (ide) ecn. Accessed: 2024-10-27.

[63] Compute Express Link Consortium. Cxl 2.0 specification, 2024. Accessed: 2024-10-27.

[64] DMTF. libspdm: An open source implementation of dmtf's spdm specification. https://github.com/DMTF/libspdm. Accessed: 2024-12-09.

[65] NVIDIA Corporation. libspdm in nvidia open gpu kernel modules. https://github.com/NVIDIA/open-gpu-kernel-modules/tree/main/src/nvidia/src/libraries/libspdm. Accessed: 2024-12-09.

[66] Gurunath Kadam, Danfeng Zhang, and Adwait Jog. Bcoal: Bucketing-based memory coalescing for efficient and secure gpus. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 570–581. IEEE, 2020.

[67] Gurunath Kadam, Danfeng Zhang, and Adwait Jog. Rcoal: mitigating gpu timing attack via subwarp-based randomized coalescing techniques. In *2018 IEEE international symposium on high performance computer architecture (HPCA)*, pages 156–167. IEEE, 2018.

[68] Michael Gruhn and Tilo Müller. On the practicability of cold boot attacks. In *2013 International Conference on Availability, Reliability and Security*, pages 390–397, 2013.

[69] Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Reetuparna Das, and Todd Austin. Cold boot attacks are still hot: Security analysis of memory scramblers in modern processors. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 313–324. IEEE, 2017.

[70] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, May 2009.

[71] Dayeol Lee, Dongha Jung, Ian T Fang, Chia-Che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[72] Yungang Bao, Mingyu Chen, Yuan Ruan, Li Liu, Jianping Fan, Qingbo Yuan, Bo Song, and Jianwei Xu. Hmtt: a platform independent full-system memory trace monitoring system. In *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 229–240, 2008.

[73] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 183–196, 2007.

[74] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. Improving cost, performance, and security of memory encryption and authentication. *ACM SIGARCH Computer Architecture News*, 34(2):179–190, 2006.

[75] Peng Gu, Shuangchen Li, Dylan Stow, Russell Barnes, Liu Liu, Yuan Xie, and Eren Kursun. Leveraging 3d technologies for hardware security: Opportunities and challenges. In *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*, pages 347–352, 2016.

[76] Akhila Gundu, Ali Shafiee Ardestani, Manjunath Shevgoor, and Rajeev Balasubramonian. A case for near data security. In *Workshop on Near-Data Processing*, 2014.

[77] Jonathan Valamehr, Mohit Tiwari, Timothy Sherwood, Ryan Kastner, Ted Huffmire, Cynthia Irvine, and Timothy Levin. Hardware assistance for trustworthy systems through 3-d integration. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 199–210, 2010.

[78] Semiconductor Engineering. Designing for security. https://semiengineering.com/designing-for-security-2/. Accessed: 2024-12-09.

[79] Jong Chern Lee, Jihwan Kim, Kyung Whan Kim, Young Jun Ku, Dae Suk Kim, Chunseok Jeong, Tae Sik Yun, Hongjung Kim, Ho Sung Cho, Sangmuk Oh, et al. High bandwidth memory (hbm) with tsv technique. In *2016 International SoC Design Conference (ISOCC)*, pages 181–182. IEEE, 2016.

[80] JEDEC Solid State Technology Association. High bandwidth memory (hbm3) dram. https://www.jedec.org/standards-documents/docs/jesd238a. Accessed: 2024-12-09.

[81] Shaizeen Aga and Satish Narayanasamy. Invisimem: Smart memory defenses for memory bus side channel. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 94–106, New York, NY, USA, 2017. Association for Computing Machinery.

[82] Juechu Dong, Jonah Rosenblum, and Satish Narayanasamy. Toleo: Scaling freshness to tera-scale memory using cxl and pim. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ASPLOS '24, page 313–328, New York, NY, USA, 2025. Association for Computing Machinery.

[83] Seonjin Na, Jungwoo Kim, Sunho Lee, and Jaehyuk Huh. Supporting secure multi-gpu computing with dynamic and batched metadata management. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 204–217, Los Alamitos, CA, USA, March 2024. IEEE Computer Society.

[84] Onur Mutlu and Jeremie S Kim. Rowhammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(8):1555–1571, 2019.

[85] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[86] Intel Corporation. 5th gen intel xeon scalable processors. https://www.intel.com/content/www/us/en/products/docs/processors/xeon/5th-gen-xeon-scalable-processors.html. Accessed: 2024-12-09.

[87] Nvidia nsight systems. https://developer.nvidia.com/nsight-systems. Accessed: 2024-12-09.

[88] Sumin Kim, Seunghwan Oh, and Youngmin Yi. Minimizing gpu kernel launch overhead in deep learning inference on mobile gpus. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, pages 57–63, 2021.

[89] Sooraj Puthoor, Xulong Tang, Joseph Gross, and Bradford M Beckmann. Oversubscribed command queues in gpus. In *Proceedings of the 11th Workshop on General Purpose GPUs*, pages 50–60, 2018.

[90] Mark Harris. How to optimize data transfers in cuda c/c++. https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/. NVIDIA Blog, Accessed: 2024-12-09.

[91] Intel Corporation. Intel® TDX Connect TEE-IO Device Guide. https://www.intel.com/content/www/us/en/content-details/772642/intel-tdx-connect-tee-io-device-guide.html. Accessed: 2024-12-09.

[92] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.

[93] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 innovative parallel computing (InPar)*, pages 1–10. Ieee, 2012.

[94] Lifeng Nai, Yinglong Xia, Ilie G Tanase, Hyesoon Kim, and Ching-Yung Lin. Graphbig: understanding graph computing in the context of industrial solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[95] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. Tigr: Transforming irregular graphs for gpu-friendly graph processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 622–636, New York, NY, USA, 2018. Association for Computing Machinery.

[96] Brendan Gregg. Flame graphs. http://www.brendangregg.com/flamegraphs.html. Accessed: 2024-12-09.

[97] Linux Kernel Community. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. Accessed: 2024-12-09.

[98] NVIDIA Corporation. Parallel thread execution isa. https://docs.nvidia.com/cuda/parallel-thread-execution/. Accessed: 2024-12-09.

[99] Lingqi Zhang, Mohamed Wahib, and Satoshi Matsuoka. Understanding the overheads of launching cuda kernels. *ICPP19*, pages 5–8, 2019.

[100] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. Automatic horizontal fusion for gpu kernels. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 14–27. IEEE, 2022.

[101] Size Zheng, Siyuan Chen, Peidi Song, Renze Chen, Xiuhong Li, Shengen Yan, Dahua Lin, Jingwen Leng, and Yun Liang. Chimera: An analytical optimizing framework for effective compute-intensive operators fusion. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1113–1126. IEEE, 2023.

[102] Bo Qiao, Oliver Reiche, Frank Hannig, and Jïrgen Teich. From loop fusion to kernel fusion: A domain-specific approach to locality optimization. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 242–253. IEEE, 2019.

[103] Guibin Wang, YiSong Lin, and Wei Yi. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 344–350. IEEE, 2010.

[104] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P Sadayappan. On optimizing machine learning workloads via kernel fusion. *ACM SIGPLAN Notices*, 50(8):173–182, 2015.

[105] Changho Hwang, KyoungSoo Park, Ran Shu, Xinyuan Qu, Peng Cheng, and Yongqiang Xiong. {ARK}:{GPU-driven} code execution for distributed deep learning. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 87–101, 2023.

[106] NVIDIA Developer Blog. Cuda graphs. https://developer.nvidia.com/blog/cuda-graphs/, 2025. Accessed: 2025-03-12.

[107] Jonah Ekelund, Stefano Markidis, and Ivy Peng. Boosting performance of iterative applications on gpus: Kernel batching with cuda graphs. *arXiv preprint arXiv:2501.09398*, 2025.

[108] Mark Harris. How to overlap data transfers in cuda c/c++. https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/. NVIDIA Blog, Accessed: 2024-12-09.

[109] Zhen Zheng, Chanyoung Oh, Jidong Zhai, Xipeng Shen, Youngmin Yi, and Wenguang Chen. Versapipe: a versatile programming framework for pipelined computing on gpu. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 587–599, 2017.

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

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

[112] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[113] Forrest N Iandola. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[114] Fei Wang, Mengqing Jiang, Chen Qian, Shuo Yang, Cheng Li, Honggang Zhang, Xiaogang Wang, and Xiaoou Tang. Residual attention network for image classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3156–3164, 2017.

[115] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.

[116] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[117] PyTorch Contributors. Automatic mixed precision (amp). https://pytorch.org/docs/stable/amp.html. Accessed: 2024-12-09.

[118] NVIDIA Corporation. Automatic mixed precision for deep learning. https://developer.nvidia.com/automatic-mixed-precision. Accessed: 2024-12-09.

[119] Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[120] AI@Meta. Llama 3 model card. 2024.

[121] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in*

Natural Language Processing: System Demonstrations, pages 38–45, Online, October 2020. Association for Computational Linguistics.
[122] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.
[123] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100, 2024.
[124] NVIDIA Corporation. Tensorrt model optimizer guide: Choosing quantization methods. https://nvidia.github.io/TensorRT-Model-Optimizer/guides/_choosing_quant_methods.html, 2025. Accessed: 2025-4-7.
[125] Yifan Tan and Zeyu Mi. Performance analysis and optimization of nvidia h100 confidential computing for ai workloads. In *2024 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 1426–1432, 2024.
[126] Yongqin Wang, Rachit Rajat, Jonghyun Lee, Tingting Tang, and Murali Annavaram. Fastrack: Fast io for secure ml using gpu tees. *arXiv preprint arXiv:2410.15240*, 2024.
[127] Ardhi Wiratama Baskara Yudha, Jake Meyer, Shougang Yuan, Huiyang Zhou, and Yan Solihin. Lite: a low-cost practical inter-operable gpu tee. In *Proceedings of the 36th ACM International Conference on Supercomputing*, ICS '22, New York, NY, USA, 2022. Association for Computing Machinery.
[128] Luyi Kang, Yuqi Xue, Weiwei Jia, Xiaohao Wang, Jongryool Kim, Changhwan Youn, Myeong Joon Kang, Hyung Jin Lim, Bruce Jacob, and Jian Huang. Iceclave: A trusted execution environment for in-storage computing. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 199–211, 2021.
[129] Md Hafizul Islam Chowdhuryy, Myoungsoo Jung, Fan Yao, and Amro Awad. D-shield: Enabling processor-side encryption and integrity verification for secure nvme drives. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 908–921. IEEE, 2023.
[130] Zaid Qureshi, Vikram Sharma Mailthody, Isaac Gelado, Seungwon Min, Amna Masood, Jeongmin Park, Jinjun Xiong, Chris J Newburn, Dmitri Vainbrand, I-Hsin Chung, et al. Gpu-initiated on-demand high-throughput storage access in the bam system architecture. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 325–339, 2023.
[131] Yang Zhou, Mark Wilkening, James Mickens, and Minlan Yu. Smartnic security isolation in the cloud with s-nic. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 851–869, 2024.
[132] Seonjin Na, Jungwoo Kim, Sunho Lee, and Jaehyuk Huh. Supporting secure multi-gpu computing with dynamic and batched metadata management. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 204–217. IEEE, 2024.

Appendix

A. *Abstract*

Our artifact provides the necessary scripts, processed data, and source code to reproduce Figures 3 through 13 in the paper. Additionally, it includes scripts for system configuration, building a TDX-patched kernel and launching a TD.

B. *Artifact Check-List (Meta-Information)*

- **Compilation:** CUDA, Python
- **Run-time Environment:** Tested on x86
- **Output:** Profile reports and generated figures
- **Disk Space Requirement:** Approximately 500GB
- **Workflow Preparation Time:** Approximately three hours
- **Experiment Execution Time:** Approximately one week
- **Publicly Available?:** Yes
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.15226687

C. *Description*

1) Accessing the Artifact: The artifact can be accessed by downloading the code archive from the provided URL.
https://doi.org/10.5281/zenodo.15226687
https://github.com/insight-cal-uva/hcc-ispass25-artifact

2) Hardware Dependencies: The experiments were conducted on the following hardware setup:

- Intel Xeon 6530 Gold CPU
- NVIDIA H100 GPU
- Supermicro SYS-421GE-TNRT3 Server

D. *Installation*

To set up the environment, download the code archive and assume you are already in the `$ROOT` directory, which is the location where the archive is stored. All operations require `sudo`.

1. Build the TDX-patched kernel: Please follow [13] to build the Linux kernel. The TDX tools are included in the code archive at `tdx-tools-2023ww15/`.
2. Verify the TDX status run:
```
hcc-scripts/tdx_check.sh
```
3. Disable Hyperthreading run:
```
hcc-scripts/hyperthreading.sh 0
```
4. Disable NUMA balancing run:
```
hcc-scripts/numa_balance.sh 0
```
5. Lock the CPU frequency run:
```
hcc-scripts/cpu_freq_lock_full.sh
```
6. Set the GPU to CC mode. Update the GPU ID in the script before execution:
```
hcc-scripts/gpu-admin-tools/cc_on_1.sh
```
7. Unbind the GPU PCI driver. Update the GPU ID (can be get from `nvidia-smi`) in the script before execution:
```
hcc-scripts/unbind_pci_1.sh
```
8. Resize the QEMU guest image to 500GB following [13].
9. Launch the TD:
```
cd hcc-scripts/tdx-tools-2023ww15
./qemu_launch_cvmgpu.sh
```
10. Copy the entire code archive into the TD and install CUDA inside TD:

```
hcc-scripts/td_guest_cuda.sh
```

11. Follow the instructions in the README file provided in the code archive to execute performance evaluation experiments.

E. Experiment Workflow

To generate a specific figure (e.g., Fig. x), execute:

```
cd figx
python figx.py
```

F. Evaluation and Expected Results

The generated figures will be stored in:

```
figure/
```

These results should match those presented in the paper.

G. Methodology

The submission, reviewing, and artifact badging methodology follows:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae