AI Workloads Performance with Safe IO Memory Protection

Mayank Bhatia Siyuan Chai Tyler Gu Shengjie Ma Xiaojuan Ma
Author list in alphabetical order

1 Introduction

As the growth of LLM workloads significantly outpaces the growth of GPU memory capacity, modern model serving frameworks consider CPU memory as a larger but slower memory layer. The serving frameworks take advantage of the large CPU memory pool in many different ways, including: (1) offloading part of the model weights to support large models [1, 3, 18], (2) offloading part of the KV cache for longer sequence length [11, 13, 18], (3) offloading the KV cache of external knowledge databases for RAG models [7, 12]. The frameworks need to migrate memory in between CPU and GPU: it copies memory to GPU before computation starts and moves unused memory to CPU when GPU memory is limited. The memory movement requires direct memory access (DMA) to read/write host memory.

Modern CPUs employ the Input-Output Memory Management Unit (IOMMU) to protect host memory from DMA accesses of malicious or buggy IO devices and drivers. IOMMU enforces protection through its translation process. With IOMMU enabled, the operating systems allocate IO virtual addresses (IOVA) which will be used by devices to initiate DMA to host memory. For each DMA, IOMMU, sitting at the root of PCIe bus, uses IO page table to translate IOVA to the physical address of host memory. The physical address points to data that the devices are allowed to access. The translation results also contain access permissions of given IOVAs. If translation or permission checks fail, DMA accesses will not happen. Similar to TLB which caches MMU's translation result, IOTLB is deployed to cached IOMMU's translation result. In other words, every IOTLB miss leads to an IO page table walk.

However, memory safety does not come free. Recent research has found that providing strict memory protections with IOMMU naively may degrade the performance of the application by up to 60% [15, 17]. To provide the strongest safety property (commonly referred as strict mode), OS unmaps IOVA and invalidates its IOTLB immediately after usage of each IO virtual address (IOVA), Such an operation leads to non-trivial performance overhead and may degrade application performance significantly. The performance impact is exacerbated in virtualized cases, since nested IO page table walks are more expensive, and frequent VM exits may also happen. Applications such as Memcached and Nginx suffer from up to 97% throughput degradation while enforcing strict safety properties [4, 19].

Existing studies focus on performance of the network workload, where the NIC performs frequent DMA operations. However, the IO memory protection's performance impact for AI workloads on GPU is poorly understood. We suspect that AI workloads which offload memory to the CPU or need to frequently exchange data between the CPU and GPU will suffer from the overhead of IOMMU protection. To our best knowledge, no existing literature measures the overhead of IOMMU protection for AI workloads on GPU.

We tested three types of workloads that may be impacted IOMMU: KV cache offloading, model weights offloading, and RAG. We found that KV cache offloading suffers from up to 10% decode stage throughput drop when IOMMU is turned on. The prefill stage is not impacted. We also conclude that IOMMU has little impact on model weights offloading and RAG.

We profiled the performance of `cudaMemcpy`, the cuda runtime API that initiates CPU-to-GPU memory transfer. We found that IOMMU introduced up to 17.5% overhead at region size in between 2^{12} to 2^{14} region sizes. The impact of IOMMU reduces at small or larger regions. We also traced the GPU kernel module [14] to pinpoint the data path for CPU-GPU data migration.

To this end, the primary goal of our work is to:

- Identifying which types of AI workload will be affected by IOMMU and quantify their impact.
- Quantify how `cudaMemcpy` will be affected by IOMMU.
- Understand the data path for CPU-GPU data migration.

2 Background and Motivation

Offloading to CPU is necessary Offloading to the CPU is necessary because GPU memory alone cannot store both the static model weights and the dynamic KV cache required for modern LLM inference. The rapid growth of model sizes and context windows quickly outpaces available GPU memory [11, 13, 18]. Moreover, consumer-grade and edge devices (designed for cost and power efficiency) come with significantly less GPU memory, making CPU offloading essential for hosting LLMs [1, 11, 13, 18].

The GPU memory wall is illustrated by FlexGen [18] in the context of training a 175B parameter model. GPT-175B requires 325GB just to load its weights, meaning that an all-GPU system would need at least five A100 GPUs (80GB each). To overcome this expensive requirement, FlexGen offloads both model weights and KV cache data across the GPU, CPU, and disk. With this approach, GPT-175B can be run on a single consumer-grade NVIDIA T4 (16GB), achieving a 25× reduction in expensive GPU memory usage. Similarly, Superpipeline [1] swaps entire model layers between the CPU and GPU, allowing start-of-the-art models like Stable Diffusion to run on graphics cards with as little as 24GB of GPU RAM.

In addition to addressing large model sizes, CPU offloading is particularly motivated by the rapidly growing KV cache, which scales with the sequence length and batch size. HeadInfer [13] offloads all but one head of the KV cache to CPU memory, enabling 4-million-token inference with an 8B model on a single consumer GPU with 24GB memory (e.g., NVIDIA RTX 4090). Similarly, InfiniGen [11] offloads most KV tokens and selectively retains important KV tokens from multiple heads based on a lightweight rehearsal of the next attention computation. InfiniGen preserves model performance in 1-million-token inference with an 8B model on a single consumer GPU with 48GB memory.

These offloading strategies require frequent CPU–GPU data exchanges. For example, Superpipeline is three times slower than an all-GPU baseline due to the additional data transfers, and HeadInfer saturates the PCIe bus throughout its execution. In every case, extra CPU–GPU traffic is traded for significantly lower GPU memory usage.

IOMMU protection and its performance overhead While CPU offloading technique is becoming more and more popular, the safety concern for DMA operations, the most important technique to enable data migrations between CPU and devices, is receiving more attention. To protect against malicious or buggy devices and device drivers, modern servers introduce IOMMU. Without IOMMU, devices directly access CPU physical memory, which may lead to information leak or system corruption. IOMMU protects memory by forcing the device to use IO virtual addresses (IOVA) and checking the access permission for the hardware's DMA access [9]. The IOVA is allocated by the kernel; it is mapped to specific physical addresses through the IOMMU page table.

The safety benefits provided by IOMMU may come with performance overhead. As we discussed, in bare metal setup, IOMMU may degrade performance by up to 60%. The performance drop is primarily attributed to IOMMU translation overhead. The severest performance drop happens under *strict* IOMMU protection policy where IOTLB is flusehd immediately after DMA transactions finish. To increase the reusability of IOTLB entries, linux's default IOMMU setup runs with *lazy* mode: IOTLB invalidation requests are queued until it reaches a max limit of 256 or every 10ms.

While many works characterize performance impact of IOMMU on network workloads on NICs, we find little literature talks about how AI workloads on GPU behave with different IOMMU setups. The complication of IOMMU may directly impact DMA transaction rates.

3 Performance Measurement Setup

We plan to compare the performance of the following microbenchmarks and AI workloads on the following setups.

3.1 Workloads

Weights offloading llama.cpp [3]
llama.cpp is an LLM inference framework that supports a wide range of hardware, from edge devices to the cloud. It has an option to offload some model layers to CPU. Specifically, higher layers reside on the GPU while lower layers are offloaded to CPU memory. The computation of CPU layers is also executed by the CPU. Once computed, the intermediate activations are transferred from CPU memory to GPU memory to be processed by GPU layers.

llama.cpp comes with three types of benchmarking workloads: prompt process (pp), text generation (tg), and prompt process + text generation (pg).

KV cache offloading FlexGen [18] and vLLM [10]
KV cache improves LLM inference performance by storing all preceding tokens' keys and values in memory to avoid redundant computation. However, the KV cache scales with the output sequence length and often consumes even more memory capacity than the model weight. To alleviate the growing issue of the KV cache size due to the demand for longer sequence lengths, [11, 18] proposed to offload the KV cache to CPU memory.

FlexGen [18] optimizes the offloading strategies by considering computation schedule, tensor placement, computation delegation, as well as compressing the model weights and KV cache. vLLM employs the PagedAttention attention algorithm [10] which divides the request's KV cache into blocks, each of which can contain the attention keys and values of a fixed number of tokens. vLLM offloads the KV cache to CPU memory when GPU does not have sufficient memory.

RAG Retrieval-Augmented Generation (RAG [7, 12, 16]) enhances large language models (LLMs) by incorporating external knowledge retrieved before generating a response. When a user inputs a query, the system searches a database (e.g., a vendor store) for relevant context and prepends the retrieved text to the query, forming the full input for the LLM. The model then computes key (K) and value (V) matrices for this extended input.

RAG, however, introduces CPU-GPU data transfers as the tokenized external context needs to be moved to the GPU for integration into the LLM's processing pipeline. To quantitatively assess the impact of this CPU-GPU input/output (I/O) overhead introduced by RAG, we adopted a synthetic RAG dataset, musique_s, provided by CacheBlend [20]. Using this dataset, we benchmarked the performance of the Mistral 7B model [2, 8] to specifically analyze the latency associated with these data transfers within a RAG-based inference scenario.

Micro benchmarks on `cudaMemcpy` AI workloads copy data between CPU and GPU devices with CUDA API call `cudaMemcpy`. It copies data synchronously by issuing DMAs with specified directions . When IOMMU is turned off, the DMA operates on physical addresses directly. When IOMMU is turned on, the DMA operates on IOVA pre-allocated by CUDA driver and pays overhead of IOMMU translations. Since `cudaMemcpy` is a synchronous API, most AI workloads choose its asynchronous version `cudaMemcpyAsync` to achieve better performance. `cudaMemcpyAsync` also issues DMA requests, so it will also be impacted by IOMMU. In this paper, we present our results to investigate both IOMMU's impact on both synchronous and asynchronous `cudaMemcpy`.

3.2 Environments

We focus on consumer-grade GPU with limited GPU memory. Under such a scenario, offloading to CPU becomes a natural resort. We have two setups.

- **P100** We use the Cloudlab machine with P100 GPU (12 GB memory). It comes with CPU memory of 192GB. The CPU is Intel Xeon Silver 4114.
- **GTX 5080** The cloudlab machine has limited availability; plus, the P100 GPU is too old to support open-source GPU kernel modules [14]. We assembled a workstation by ourselves. It has a GeForce RTX 5080 GPU, Intel i7-14700F CPU, and 128GB memory. The CPU and GPU are connected with PCIE 5.0.

We measured the performance using **GTX 5080** unless particularly noted since it supports the open source kernel modules.

- iommu=off
  No protection. Devices access physical memory directly. Kernel functions handles mapping request by directly returning the physical addresses.
- iommu=on, strict invalidation policy
  For strict policy, IOTLB is invalidated immediately after each DMA access finishes. It enforces the strictest safety guarantee while leading to the most significant performance impact. In this paper, we focus on strict policy.
- iommu=on, lazy invalidation policy
  For lazy policy, the default of Linux, the IOTLB invalidation request is submitted to an invalidation queue; the IOTLB is flushed periodically or whenever the queue is full. The lazy policy leads to larger chances of IOTLB reuse at the cost of a potentially longer attack window.

4 Measurement Results

We found that not all AI workloads will be affected by IOMMU. Workloads like model weight modeling and RAG are almost not influenced by IOMMU. However, workloads like KV Cache offloading will be significantly influenced by IOMMU policy. We found KV cache offloading will suffer from throughput drop of 6-11% when IOMMU is turned on.

4.1 Workloads with substantial impact

4.1.1 KV Cache Offloading with FlexGen

We quantitatively evaluated the influence of different IOMMU policies on the KV cache offloading system by analyzing the throughput of the prefill and decode phases. We ran Flexgen with facebook/opt-1.3b model [5, 21] with both prefill and token generation length of 1024. We tested the worst case scenario with 100% KV Cache offloaded to CPU, and 100% model weights plus activation on GPU. We found that when turned on IOMMU with strict policy, decode throughput dropped by 9.9% compared to when IOMMU is turned off. In contrast,

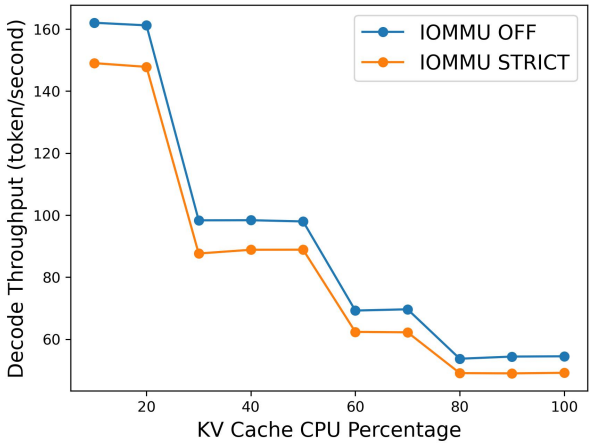

Figure 1: Decode throughput for different KV cache offloading percentages.

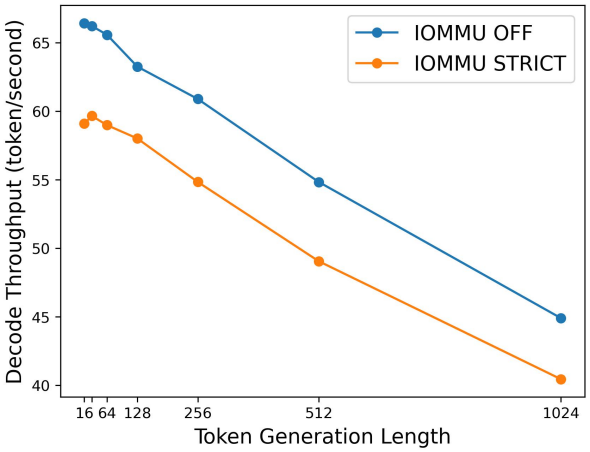

Figure 2: Decode throughput vs. token generation length

prefill throughput has smaller difference of 3.6% between IOMMU is strict and off. We explain the difference with different workloads characteristics of prefill and decode phases. prefill phase is computation bounded, while decode phase is memory bounded. More frequent CPU to GPU data migrations occur in deocde phase.

To further investigate this, we conducted two experiments with varying KV cache offloaded percentage and token generation length.

Impact of Varying KV Cache CPU Offloading Percentages. In this experiment, we fixed the input and output token lengths at 1024 and 512, respectively, and systematically increased the proportion of the KV cache offloaded to the CPU from 0% to 90%. As illustrated in Fig. 1 for the facebook/opt-1.3b model, the decode throughput decreased for both iommu-off and iommu-strict configurations as more KV cache was moved to the CPU. Notably, the iommu-strict policy consistently demonstrated a performance degradation compared to iommu-off across different offloading percentages. The degradation is 9.51% on average with minimum of 8.04% and maximum of 10.8%. In contrast, the impact of the IOMMU policies on the prefill phase throughput was less pronounced in our experiments. Specifically, the decode throughput under iommu-strict was either comparable to or slightly worse than iommu-off, with a performance difference of less than 4%.

Impact of Varying Output Token Generation Lengths. For each generated token, CPU-GPU I/O operations occur if the KV cache is even partially offloaded to the CPU. Consequently, longer generation lengths, requiring more frequent CPU-GPU data transfers, may reduce decoding throughput. Understanding the impact of the IOMMU policy on this relationship is crucial. As illustrated in Fig. 2 for the facebook/opt-1.3b model (with 100% KV cache offloaded to the CPU), the decode throughput for both iommu-off and iommu-strict policies decreased as the output token generation length increased. Notably, a consistent performance degradation of approximately 8.29% - 11.02%, with an average of 9.95%, in decode throughput was observed with the iommu-strict policy compared to iommu-off across different generation lengths. For the prefill phase, we observed some performance degradation ranging from 1% to 6%; however, no clear trend was discernible within our experimental results.

`CudaMemcpy` **Analysis** We profiled the performance of `CudaMemcpy` calls Flexgen workloads with Nsight Systems. We ran facebook/opt-1.3b with input token length of 1024 and generation length of 512 with 100% KV cache offloaded to CPU. We focused on Host-to-Device memory transfer since it take 99.1% of total transfer time. We showed the cumulative distribution functions (CDFs) of transfer time in Figure 3. Most memory transfers take place at a magnitude of 10^7 bytes and 10^6 nanoseconds. We found that IOMMU strict policy slowed down the memory transfer by 15% which led to the throughput drop in decode phase.

Flexgen with Larger Model To investigate the generalization of our findings, we further applied the same experimental procedures to the larger facebook/opt-6.7b model [6]. For the varying KV cache experiment, we were required to offload at least 60% of the KV cache to the CPU to avoid out-of-memory (OOM) errors. Despite this constraint, we observed a similar trend of consistent performance degradation in both sets of experiments. However, the performance gap between the iommu-off and iommu-strict policies was smaller for the 6.7 billion-parameter model—5.9%–7.0% in the KV-cache CPU offload experiment and 4.8%–9.4% in the token generation length experiment—than for the 1.3

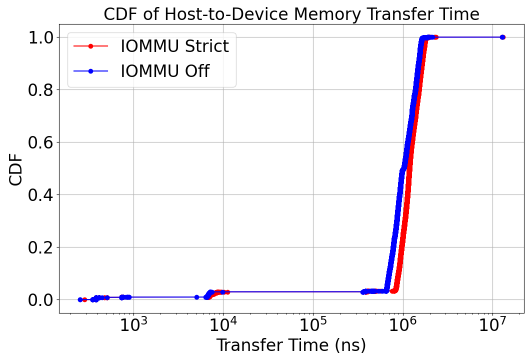

Figure 3: CDF plot for Host-to-Device memory transfer time of Flexgen running opt-1.3b.

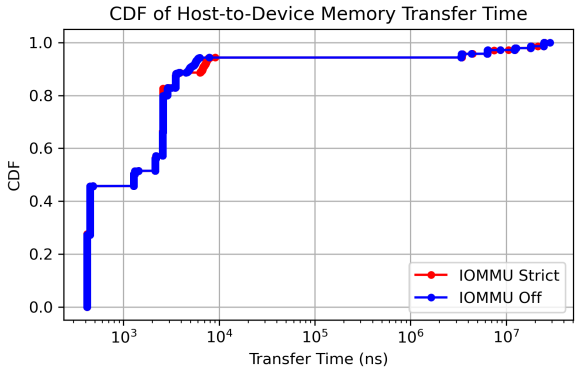

Figure 4: CDF plot for Host-to-Device memory transfer time of Llama.cpp running Phi-4.

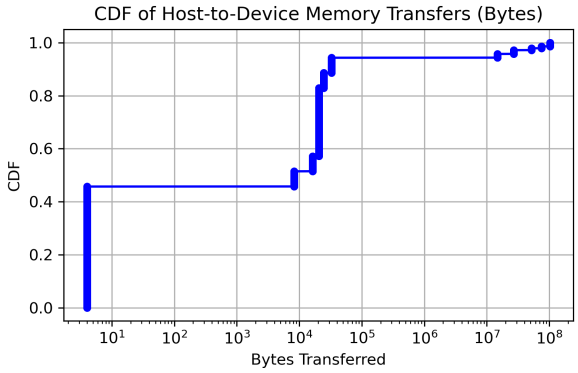

Figure 5: CDF plot for Host-to-Device memory transfer size of Llama.cpp running Phi-4.

billion-parameter model, which averaged at 10% in both experiments.

4.1.2 KV Cache Offloading with vLLM

We further evaluated the impact of IOMMU policies on the state-of-the-art serving framework, vLLM. We used the Llama-3.2-3B model with the max model length of 4800. We configured the vLLM to use the CPU memory as the swap space, so that it offloads the KV cache to CPU memory when GPU memory is not enough. We used the `sharegpt` dataset as the serving benchmark. The measurement is run on the GTX 5080 with 70% GPU memory utilization.

We found that the strict IOMMU policy increases the inter-token latency by 2.84% (from 34.96ms to 35.95ms) comparing to IOMMU set to off. However, the IOMMU policies do not have significant impact the time-to-first-token latency which is consistent with our finding that prefill stage performance is not impacted by IOMMU. Our future plan is to further investigate the impact of IOMMU policies on vLLM with different models, model length, output token length, and the amount of GPU memory available.

4.2 Workloads without substantial impact

4.2.1 Model Weights Offloading

To evaluate the impact of IOMMU policy on the model weights offloading strategy in `llama.cpp`, we benchmarked the throughput of the prefill and decode stages. Using the `llama-bench` tool provided with `llama.cpp`, we profiled an 8-bit quantization of Llama-3.1-8B and a 4-bit quantization of Phi-4-14B. For both models, one layer is offloaded to the CPU, while the remaining layers reside on the GPU. All experiments were conducted on the P100 setup.

Table 1 reports the mean and standard deviation across five runs with a prompt length of 512 and 128 generated tokens. For both models and both stages, the throughput differences fall within the standard deviation, indicating that the IOMMU policy has negligible impact.

To understand the underlying reasons for this observation, we profiled Host-to-Device memory transfers during the decode stage of Phi-4 using Nsight Systems. The cumulative distribution functions (CDFs) of transfer time and transfer size are shown in Figure 4 and Figure 5, respectively. Transfer events can be categorized into two groups based on size: those below 10^5 bytes, corresponding primarily to activation vectors transferred in each decoding iteration, and those above 10^7 bytes, corresponding mainly to model weights transferred once at the start of benchmarking. The primary difference between the CDFs arises from transfers around 10^4 bytes, which are associated with activation data.

A deeper inspection of the profiling data reveals that strict IOMMU configuration slows down model weight

Model	Stage	#Tokens	IOMMU=off	IOMMU=on, strict
Llama-3.1-8B	prefill	512	468.95 ± 0.90	468.70 ± 1.31
	decode	128	21.68 ± 0.19	21.77 ± 0.18
Phi-4-14B	prefill	512	210.31 ± 0.33	210.32 ± 0.33
	decode	128	17.22 ± 0.13	17.20 ± 0.37

Table 1: Throughput (tokens/s) comparison under different IOMMU configurations. Throughput is displayed in the format mean ± std.

transfers by 2% and activation transfers by 6%. However, weight transfers occur only once during setup and do not affect inference throughput. In the model weight offloading strategy used by llama.cpp, GPU-resident layers are transferred from CPU to GPU memory at initialization. During inference, CPU-resident layers remain in CPU memory and are computed on the CPU, while GPU-resident layers remain on the GPU and are processed there. Activation transfers take approximately 6 μs per token, accounting for only 0.01% of the total decoding time. This impact is negligible.

In conclusion, IOMMU configuration has no substantial effect on the performance of model weights offloading in `llama.cpp`. Therefore, we recommend enabling IOMMU to benefit from its memory protection features at little or no performance cost.

4.2.2 RAG

To evaluate the impact of the IOMMU policy on RAG systems, we assessed the Mistral-7B-Instruct-v0.1 model [2] on the musique_s.json dataset from CacheBlend [20]. This dataset comprises 150 queries with input tensor sizes ranging from 47,824 to 63,184 bytes.

The total inference time is dominated by the token generation process, which operates on the scale of seconds, while the memory transfer times occur in the scale of hundreds of microseconds. Consequently, the impact of different IOMMU policies on the end-to-end inference time is minimal (<0.1%). However, the cumulative distribution function (CDF) of memory transfer times (Fig. 6) reveals a more nuanced effect. The intersection of the CDF curves for iommu-off and iommu-strict indicates a non-trivial influence of the IOMMU policy on memory transfer latency, despite its limited impact on the overall inference duration.

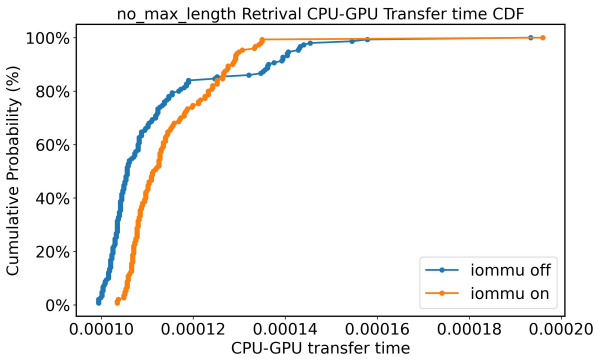

Figure 6: CDF plots for RAG system with iommu-off and iommu-on (strict).

4.3 Microbenchmarks

In this section, we discuss four microbenchmarks used to understand the performance impact of the IOMMU in controlled environments.

- **Repeated Single Region Transfer** A single region of CPU memory is repeatedly copied to a single region of GPU memory 256 times. The entire process is looped over 100 times for consistency with the following microbenchmarks.
- **Sequential Contiguous Transfers** 256 contiguous regions of CPU memory are sequentially copied to a single region of GPU memory. The entire process is looped over 100 times to capture the impact of IOTLB speeding up translation.
- **Random CPU Transfers** 192 fragmented regions of CPU memory are randomly copied to a single region of GPU memory. The fragmentation is obtained by randomly selecting 75% of 256 contiguous regions to copy from in a random order. The entire process is looped over 100 times, updating the random selection in each iteration, to capture the impact of IOTLB speeding up translation.
- **Fully Random Transfers** 192 fragmented regions of CPU memory are randomly copied to 192 fragmented regions of GPU memory. The entire process is looped over 100 times.

Latency Model Unfortunately, existing profiling tools cannot profile the breakdown inside `cudaMemcpy` operations since the cuda API implementation is closed source.

To help us better understand the measured results, we model latency in each microbenchmark as follows. When the IOMMU is enabled, then the total latency $T_{\text{total,on}}$ is given by the sum of three components:

$$T_{\text{total,on}} = T_{\text{DMA}} + T_{\text{other}} + T_{\text{IOMMU}}$$

- $T_{\textbf{DMA}}$: Latency from direct memory access between GPU and CPU memory.
- $T_{\textbf{other}}$: Mixed overheads unrelated to DMA or IOMMU operations, such as kernel launches CUDA runtime overhead.
- $T_{\textbf{IOMMU}}$: Latency introduced from IOMMU operations such as address translations and IOTLB lookups.

On the other hand, when the IOMMU is disabled, then the total latency is simplified to:

$$T_{\text{total,off}} = T_{\text{DMA}} + T_{\text{other}}$$

To isolate the performance impact of the IOMMU, we compare $T_{\text{total,off}}$ with $T_{\text{total,on}}$. In additon to comparing the raw values, we also look at the latency ratio:

$$\text{IOMMU Overhead } \% = (\frac{T_{\text{total,on}}}{T_{\text{total,off}}} - 1) \times 100\%$$

We evaluated the four microbenchmarks for both synchronous `cudaMemcpy()` and asynchronous `cudaMemcpyAsync()`.

4.3.1 `cudaMemcpy` results

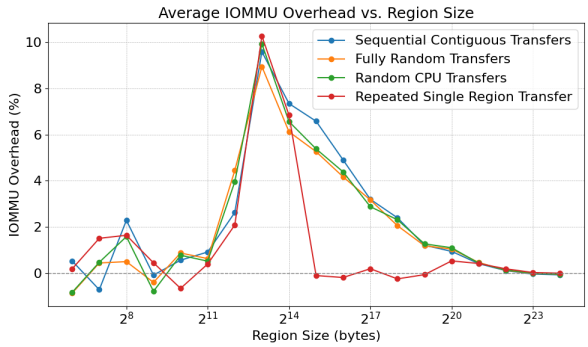

Figure 7: IOMMU Overhead for `CudaMemcpy` with Different Region Sizes

Figure 7 shows the IOMMU overhead, normalized to the memory copy time of the IOMMU-off case. For all the access patterns, the plots exhibit an "arch" shape: the IOMMU overhead remains consistently low for small `cudaMemcpy` operations involving small regions, peaks as region sizes grow beyond a page, and gradually declines as the region size continues to increase.

Small region sizes (< 2^{12} bytes) The IOMMU overhead hovers around 1%. This is expected, as the small working set does not stress the IOTLB, resulting in minimal translation overhead.

Medium region sizes (2^{12}–2^{13} bytes) The overhead rises sharply. At this scale, each `cudaMemcpy` likely incurs at least one IOMMU translation, amplifying the T_{IOMMU} the region size grows. Meanwhile, the T_{DMA} and T_{other} stay relatively stable. In Figure 8, we show the memory transfer time for different region size normalized to the time taken by smallest region (2^6) bytes when IOMMU=off. Since different access patterns show the same trend, we show the plot from "Sequential Contiguous Transfers" as a example With IOMMU = off, the memory transfer time for the 2^{13} region size is only 1.25x of the memory transfer time for 2^6 region size.

Large region sizes (>= 2^{14} bytes) The overhead gradually decreases. With larger regions, T_{DMA} increase exponentially with region size as shown in Figure 8. The cost of IOMMU translation becomes amortized over data transfer time, leading to a lower relative overhead.

The peak in IOMMU overhead is skinniest for repeated single region transfers because this microbenchmark have the highest IOTLB cache hit rate. Note that the peak is at 2^{13}, which is the first data point greater than the single page size 2^{12}. The peak diminishes back to 0 for large region sizes because T_{DMA} begins to dominate the total latency due to larger overall memory size.

Limitation in Analysis Note that Figure 8 is combination of T_{DMA} and T_{other}. We wish to profile T_{DMA} separately but we didn't find any open tool to anlayze the pure the time for each DMA transfer. We also could have better understanding of T_{IOMMU} by profiling the IOTLB miss rates and IOMMU cache miss rates following the methodology in [17]. However, IOMMU performance registers are only avaible on server grade CPU, where our 5080 setup does not have it.

4.3.2 `cudaMemcpyAsync` results

We measured the normalized IOMMU overhead for four different access patterns over region sizes from 2^6 to 2^{24}. `cudaMemcpyAsync` copy data asynchronously in the background. We launched all 256 `cudaMemcpyAsync` calls at one time, and recorded the time taken to finish all of them as the time required to transfer. We showed our plot in Figure 9.

We found that, in general, the asynchronous results follow the same pattern of "arch" shape of the synchronous results. IOMMU overhead increases at region sizes from 2^{12} to 2^{14} since T_{DMA} is not the dominate time. However,

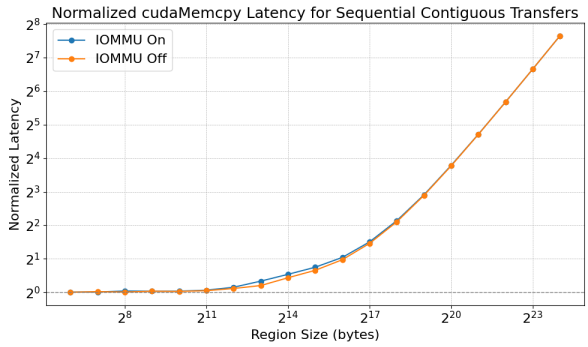

Figure 8: `CudaMemcpy` latency normalized to region size (2^6) for access pattern: Sequential Contiguous Transfers. Despite the variability shown in Figure 7, we observe minimal variability here, so we show result from one access pattern as example.

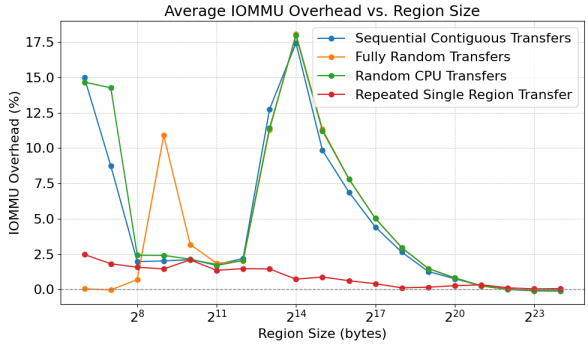

Figure 9: IOMMU Overhead for `CudaMemcpyAsync` with Different Region Sizes

as region size grows larger, T_{DMA} dominates the memory transfer time, so IOMMU overhead gradually decays.

In asynchronous case, the maximum peak has a higher IOMMU overhead of 17.5% compared to 10% in the synchronous case. This can be explained by the differen in workload pattern: asynchronous memory copy requests are submitted in burst ,but the synchronous requests are submitted one by one. In async case, IOMMU translation requests may be blocked at IOMMU hardware.

We also found that "Repeated Single Region Transfer" did not follow the "arch" shape. We are still working to check the reason One hypothesis is that bursty memory copy requests might be better handled by IOTLB where the requests have higher possibility of IOTLB hit. The other hypothesis is that the cuda runtime (currently closed source) might detect that we are copying to a GPU region from the same CPU region repeatedly, so it might optimize by only copying the final results.

Note that for small region sizes, asynchronous memcpy is far less predictable, especially the access patterns "Sequential Contiguous Transfers" and "Random CPU Transfers" at region size smaller than 2^8 bytes. We suspect that it might be caused by the synchronization overhead where these two workloads are copying to a single GPU region within one page from multiple CPU region. The cuda runtime may need more time to make synchronize the final results.

4.4 Code Path Understanding

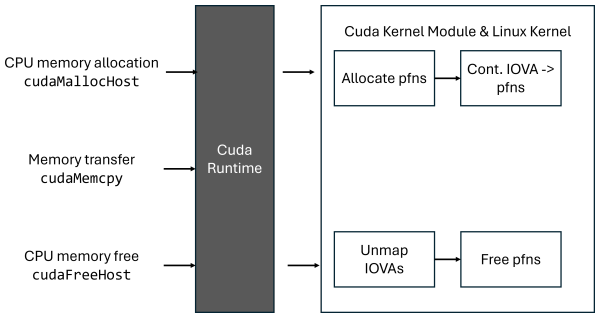

Figure 10: Code Path for CPU-GPU Communication

Figure 10 illustrates the memory allocation and deallocation code path for CUDA application that involves CPU-GPU memory communication. It contains user-level cuda source files that calls the CUDA runtime API. The CUDA runtime is closes source, but it will call the open source the Linux kernel via a CUDA kernel module.

When a CUDA application allocates CPU memory and plans to transfer the data to GPU, it requests memory via `cudaMallocHost` call. The CUDA runtime forwards this request to a kernel module. The kernel module, in collaboration with the Linux kernel, allocates physical pages (`pfns`) and sets up a contiguous IOVA (IO Virtual Address) space. The module then maps this contiguous IOVA range to the allocated `pfns`, enabling GPUs to access the host memory with IOVA via the IOMMU subsystem. Note that the physical pages may be noncontiguous, but the kernel module enforces contiguous IOVA for performance reason.

When the memory is no longer needed (e.g., via `cudaFreeHost()`), the process reverses: the IOVA mappings are freed first, followed by the release of the associated physical pages.

We emphasize that the building and removal of IOVA mappings do not happen at `cudaMemcpy` time. In fact, we did not find `cudaMemcpy` enter the kernel module. Such design avoids the overhead of entering kernel space for `cudaMemcpy`. The memory copy can be accomplished via IOVA mappings set up at allocation time, and such mapping may be reused multiple times.

5 Conclusion and Future Work

In this work, we conducted the first detailed study of the performance impact of IOMMU protections on AI workloads involving GPU–CPU memory transfers. While prior research has primarily focused on network devices, our findings highlight that certain LLM inference scenarios—especially those involving KV cache offloading—suffer up to 10% throughput degradation under strict IOMMU policies. We further identify the region sizes and transfer patterns that exacerbate IOMMU-induced latency, and profile the behavior of both synchronous and asynchronous `cudaMemcpy` operations.

Future Work. We plan to extend this work by:

- Investigating more on the impact of IOMMU on modern serving framework like vLLM.
- Profiling IOTLB miss rates and IOMMU cache miss rates to more precisely isolate the cost of translation.
- Exploring alternative designs for safe DMA that reduce IOMMU overhead, such as batching, caching optimizations, or static mappings for persistent buffers.

We hope our findings motivate further systems and architecture co-design to balance memory safety and performance in AI accelerators.

References

[1] Reza Abbasi and Sernam Lim. *Superpipeline: A Universal Approach for Reducing GPU Memory Usage in Large Models*. 2024. arXiv: 2410.08791 [cs.LG]. URL: https://arxiv.org/abs/2410.08791.

[2] Mistral AI. *Mistral-7B-Instruct-v0.1*. https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.1. Hugging Face model.

[3] Georgi Gerganov et al. *llama.cpp*. [Accessed 23-02-2025]. URL: https://github.com/ggml-org/llama.cpp.

[4] Nadav Amit et al. "vIOMMU: Efficient IOMMU Emulation". In: *2011 USENIX Annual Technical Conference (USENIX ATC 11)*. Portland, OR: USENIX Association, June 2011. URL: https://www.usenix.org/conference/usenixatc11/viommu-efficient-iommu-emulation.

[5] *facebook/opt-1.3b*. https://huggingface.co/facebook/opt-1.3b. Hugging Face model.

[6] *facebook/opt-6.7b*. https://huggingface.co/facebook/opt-6.7b. Hugging Face model.

[7] Yunfan Gao et al. "Retrieval-augmented generation for large language models: A survey". In: *arXiv preprint arXiv:2312.10997* 2 (2023).

[8] Albert Q. Jiang et al. *Mistral 7B*. 2023. arXiv: 2310.06825 [cs.CL]. URL: https://arxiv.org/abs/2310.06825.

[9] Andy Kegel et al. *VIRTUALIZING IO THROUGH THE IO MEMORY MANAGEMENT UNIT (IOMMU)*. URL: https://pages.cs.wisc.edu/~basu/isca_iommu_tutorial/IOMMU_TUTORIAL_ASPLOS_2016.pdf.

[10] Woosuk Kwon et al. "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. SOSP '23. Koblenz, Germany: Association for Computing Machinery, 2023, pp. 611–626. ISBN: 9798400702297. DOI: 10.1145/3600006.3613165. URL: https://doi.org/10.1145/3600006.3613165.

[11] Wonbeom Lee et al. *InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management*. 2024. arXiv: 2406.19707 [cs.LG]. URL: https://arxiv.org/abs/2406.19707.

[12] Patrick Lewis et al. "Retrieval-augmented generation for knowledge-intensive nlp tasks". In: *Advances in neural information processing systems* 33 (2020), pp. 9459–9474.

[13] Cheng Luo et al. *HeadInfer: Memory-Efficient LLM Inference by Head-wise Offloading*. 2025. arXiv: 2502.12574 [cs.LG]. URL: https://arxiv.org/abs/2502.12574.

[14] Nvidia. *open-gpu-kernel-modules*. URL: https://github.com/NVIDIA/open-gpu-kernel-modules.

[15] Chathura Rajapaksha et al. "IOMMU Deferred Invalidation Vulnerability: Exploit and Defense". In: *2024 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2024, pp. 1–6. DOI: 10.23919/DATE58400.2024.10546528.

[16] Ori Ram et al. "In-context retrieval-augmented language models". In: *Transactions of the Association for Computational Linguistics* 11 (2023), pp. 1316–1331.

[17] Benny Rubin et al. "Fast & Safe IO Memory Protection". In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. SOSP '24. Austin, TX, USA: Association for Computing Machinery, 2024, pp. 95–109. ISBN: 9798400712517. DOI: 10.1145/3694715.

3695943. URL: https://doi.org/10.1145/3694715.3695943.

[18] Ying Sheng et al. "FlexGen: high-throughput generative inference of large language models with a single GPU". In: *Proceedings of the 40th International Conference on Machine Learning (ICML'23)*. 2023.

[19] Kun Tian et al. "coIOMMU: A Virtual IOMMU with Cooperative DMA Buffer Tracking for Efficient Memory Management in Direct I/O". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 479–492. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/tian.

[20] Jiayi Yao et al. "CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion". In: *arXiv preprint arXiv:2405.16444* (2024).

[21] Susan Zhang et al. *OPT: Open Pre-trained Transformer Language Models*. 2022. arXiv: 2205.01068 [cs.CL]. URL: https://arxiv.org/abs/2205.01068.