Multi Model Tenancy on GPUs

Akira Van De Groenendaal[1] and Tianyou Zhang[1]

[1]Computer Science Department, Carnegie Mellon University, USA

Abstract

Let's be honest, the last time you used an LLM was probably within the last few hours (or maybe minutes), and with their growing capabilities, adoption is likely to become increasingly widespread. It follows that the capacity of LLM inference providers needs to meet this demand, but many are reportedly losing money due to the high costs of GPU infrastructure, and serving complex models often outpaces revenue. One contributing factor is the under utilization of said infrastructure. Current GPU workloads, especially LLM serving, are memory bound (Recasens et al. 2025). Recent systems papers identify scheduling, memory layout, KV-cache management, and multi-model interference as key sources of inefficiency. This motivates studying GPU multi-tenancy and workload co-scheduling to increase utilization and throughput, studying the effects of interference. In this paper, we examine the impacts of multi-tenancy on encoder and decoder models, served on an NVIDIA A100 GPU with MPS. We find that there is little to no benefits .

Keywords: GPU, LLM, Encoder, Decoder, Memory, GPU Utilization, Multi-Tenancy

Background

Large language models have become a crucial part of society for several industries and even in everyday life. There are many cases where balancing between performance and saving on costs is vital to a business, such as datacenters which are forced to balance between QoS agreements and costs. There are also scenarios of edge computing where the resources are limited, so multiple models have to be run on the same GPU. Our aim for this project is to gather data for how multi tenancy on a single GPU affects key metrics, so that users are able to optimize how they run models on their GPUs.

There are many different types of machine learning models, and for LLMs there are 2 main stream architectures: encoders and decoders. In doing so we aim to get a more accurate representation of running various types of models on GPUs. Another reason is that encoders and decoders stress different parts of the GPU resources, so we hope to see if there are optimal ways of organizing various models depending on the resource bottlenecks.

Overview for Encoders and Decoders

Encoders are bidirectional and compute all tokens at the same time. Being bidirectional means that each attention layer sees all tokens before and after the token it is being applied to. Also, the tokens

are generated in parallel, which means that the models have low arithmetic intensity. Furthermore, encoders are normally used for smaller sequences, so the compute requirements are smaller. Since the models are extremely fast during the compute portion and the memory reads and writes become a larger portion of the job time. Therefore, encoders become memory bandwidth bound.

Decoders have an encoding step first, which is memory intensive, but very quick. The main chunk of computation is in the second decode phase where tokens are generated sequentially because each token depends on the previous token. Since the tokens are generated sequentially, results the decoder models become compute bound.

As explained above encoders and decoders have different resource demands, so there is potential in different combinations of the models becoming more efficient than other pairings.

Another aspect to consider is the size of the model, or the number of parameters for each model, which affects the arithmetic intensity. In order to get a broad dataset, we used models of a few different sizes.

Another important aspect to consider is how to run multiple models on the same GPU. There is the simple method of just using multiple processes and making multiple calls to the GPU. However, this results in heavy overhead for context switching. There are a few different strategies such as Multi-instance GPU by NVIDIA, which allows users to partition the resources of the GPU. However, not all GPU's have this resource and is a fairly new tool. Also, it is limited to 7 partitions, which is smaller than what we wanted to test for some instances. There is also the decision between CUDA streams and Multi-Process Service(MPS). MPS is available on fewer devices, but it is widely used in data centers or other multi-process/application situations. Therefore, we decided that performing tests with MPS would be the best concurrency tool to use.

Methodology

Experiments were run in a cloud environment from a Python driver script. Inference "requests" were submitted locally, via Python code. In experiments with multiple models, each was managed by a fork from the central driver and synchronized via barriers where necessary. While this setup does not simulate real-world request patterns, it's effective for isolating the performance impacts of multi-tenancy.

Hardware: Experiments were conducted on a single-node setup with an NVIDIA A100 40GB SXM4 GPU.

Models: Five models were evaluated: DistilGPT2, roBERTa, Llama-3.2-3B-Instruct, and Mistral-7B-Instruct-v0.2. These were chosen to encorporate models with different architectures and parameter sizes. All models fit in the 40GB memory limit.

Workload: Models were run on fp16 precision with a fixed batch size of 8 and sequence length of 128. 100 inference iterations were run with a synthetically generated, manually seeded input that was the same between models and across experiments. 10 warmup iterations were conducted before the measured section.

Profiling: To verify conclusions, Nvidia Nsight was used to track the kernel launches of each model, as well as inspect the SM and memory utilization at any moment in time during the experiment.

Testing Protocol: First, solo benchmarks were used to get baseline performance metrics for each of the models examined, similar to getting a sequential baseline when parallelizing a program.

Multitenant experiments loaded two or more models onto the GPU, and the above workload was run on each tenant. The different models were managed by separate Python processes, forked from a driver script. They were synchronized using barriers to ensure the timed iterations started concurrently. There was no synchronization between models during the timed experiment. To collect latencies, the CUDA Events module was used, which introduces minimal overhead, but should not impact the relative performances.

For our analysis of fairness and efficiency, we compare the models between each other if they are the same model, and otherwise, we compare to the solo baseline model and compare the slowdowns.

Test Case Design: A comprehensive list of experiment configuration files can be found in the GitHub repository. To summarize, the following categories were considered:

- Each model alone for baseline
- All pairs
- Encoders only
- Decoders only
- Various combinations of both encoders and decoders

The goal of these tests was to see how the quantity and type of models, and each combination, affects performance, in order to reveal biased resource allocation, memory and compute bottlenecks, as well as general efficiency limitations. In particular, we were curious if running encoders and decoders together would reduce the perceived interference.

Metrics: For our latency timing we obtained 3 different metrics of GPU kernel execution time, GPU event time, and wall clock time for all 100 iterations. This way we would be able to see where time was being spent, and allow for a deeper analysis of why speedups occurred or didn't occur.

Results

Table 1 shows baseline latencies for each model. As expected, latencies scale increasingly with model size and parameter count.

We analyze the same mode pair experiments. The purpose of these experiments was to also es-

Note Name	Mean (ms)	std	min	max	p50	p90	p95	p99
solo_gpt2	4.989	0.912	4.503	8.905	4.653	5.404	7.703	8.543
solo_distilBERT	2.904	0.504	2.578	5.363	2.722	3.467674	4.271	4.641
solo_ROBERTA	12.089	2.230	11.178	31.079	11.470	14.215	15.329	17.053
solo_llama.32	51.283	13.395	43.192	103.770	46.151	70.2527	83.360	99.307
solo_mistral7b	102.925	19.458	90.570	158.965	93.030	143.573	147.534	152.963

Table 1: Baseline experiments and statistics for per-request latency (ms)

tablish some sort of baseline for how models with the same size and parameters would perform. The expectation was that the same model would on the same GPU would split the workload evenly, which is what we observed for most of the experiments, however, there was some interesting results for the smaller models.

We are able to see a very small level of biasing in the smaller models. For GPT2 we see in Figure 2 that one model is 11% slower for the 50th percentile metric. This is a significant margin, but with t-test and Mann-Whitney U tests, the difference was not significant.

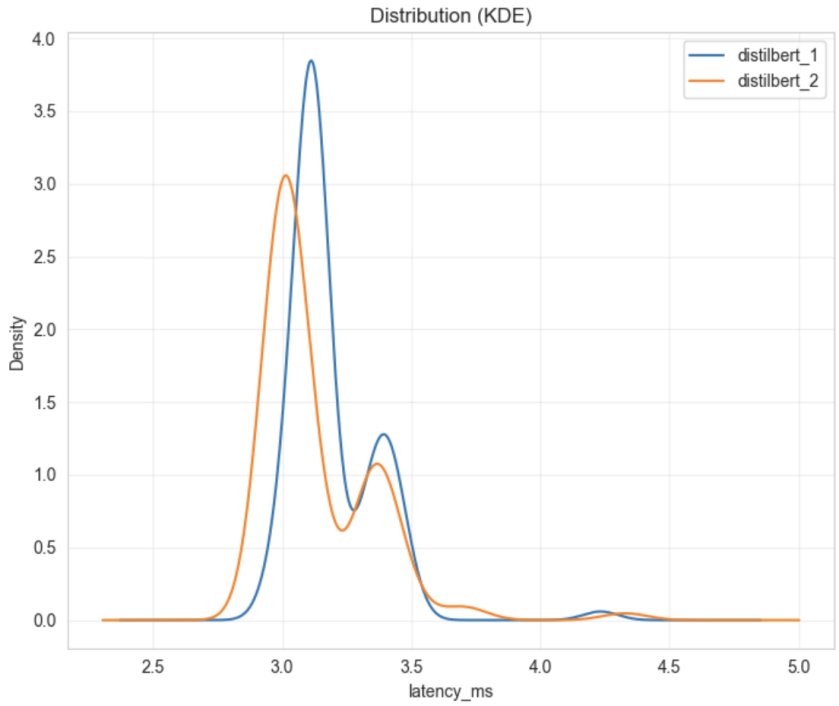

Figure 1: Pair distil BERT models Kernel Density Estimation Graph

It is also possible for the scheduling to be more fair in some scenarios such as shown with the distil bert models. However, there is still a slight preference for one model over the other. the differences could be that one is more compute bound than the other. However, the main reason

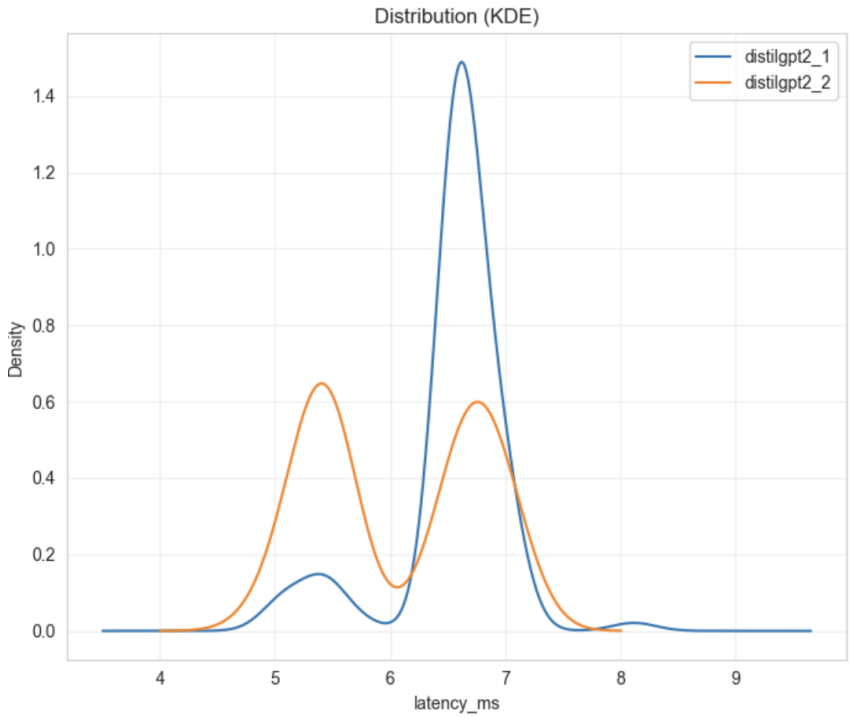

Figure 2: Pair distil GPT2 models Kernel Density Estimation Graph

that this is visible is because the models themselves are small, so being served a little later is much more significant and visible. As we can see for the larger models, there is no evident bias between the models. The bias is not significant, but this depends on the requirements.

For the experiments with varying model sizes the impact is much more evident. The smaller models are forced to share for their entire run time, which drastically reduces their performance. This is partially due to MPS scheduler which, employs time slicing. As a result, all processes that are being run together take an equal performance hit. However, due to our experiment design of using batch iterations and not some interval testing, the smaller models finish much quicker, which results in the larger models having an empty when they run. In the case of the experiment with 4 distilBERT models and 1 Mistral7b, the first 5 Mistral iterations had an average latency of 190ms, which immediately drops to 85ms when the distilBERT's finish. This discrepancy results in the larger models seemingly taking less performance hits than the smaller models.

This leads us into another interesting result, which is that the mistral model performed better when paired with other models than by itself. After the other smaller models finished, the mistral model was able to achieve lower latency than when it was on the GPU alone. On the solo experiment the model had an average latency of 103ms, and on the other experiments such as with 4 distilBERT, the mistral model had an average of 85ms. Another interesting pattern was that in the solo run, the model had many iterations where the latency would spike to 140ms. However, when mistral was ran with other models this occurrence completely disappeared. We analyzed the GPU frequency,

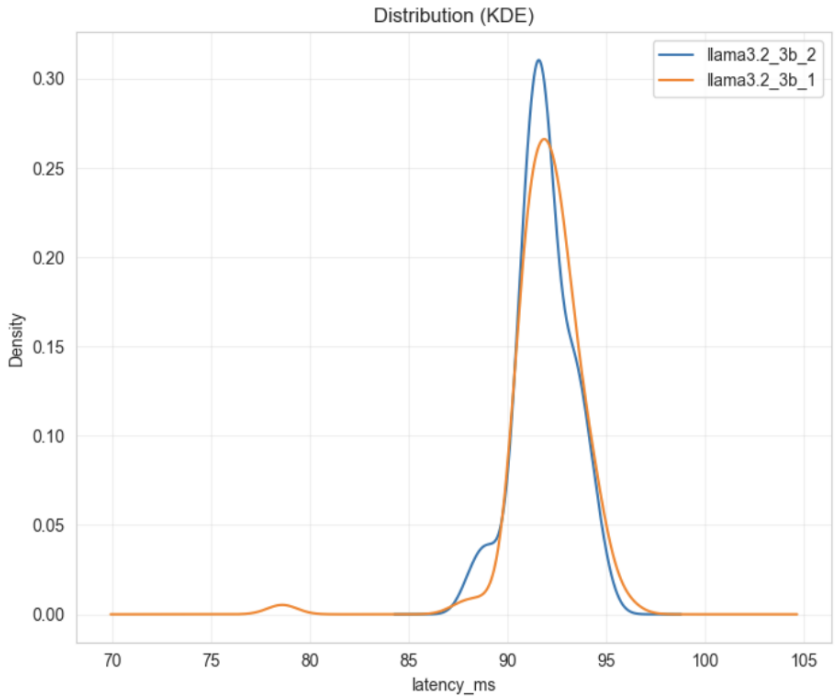

Figure 3: Pair llama3.2 models Kernel Density Estimation Graph

SM usage, and DRAM bandwidth, but everything is within a margin, that would not have impacted the performance.

Another pattern that we noticed was that SM usage and memory bandwidth not being maxed. We read in another paper Recasens et al. 2025 that memory bandwidth to be the limiting factor for model inference latency. However, we were not able to see the same results with our Nsight metrics. Our heavier experiments such as the pair of mistral models was able to achieve an average of 20% compute warps in flight, which is comparable to the values in the paper. However, our DRAM usage was much lower in all scenarios with reads capping out at 20% and writes at 10%. This combination of low compute and DRAM bandwidth leads us to believe that the GPU is not running at maximum capacity. However, the performance is decreasing with an increase in latency. Even with multiple models running we were unable to achieve a higher DRAM bandwidth usage.

We are uncertain about what could be causing limitations in our experiments since it does not seem to be any of the general bottlenecks of compute and memory. However, it could be due to cache clashing and such. There could be issues with KV caching and such. However, we do not have a deep enough understanding in ML to fully understand the layers of the models and how the compute restraints change.

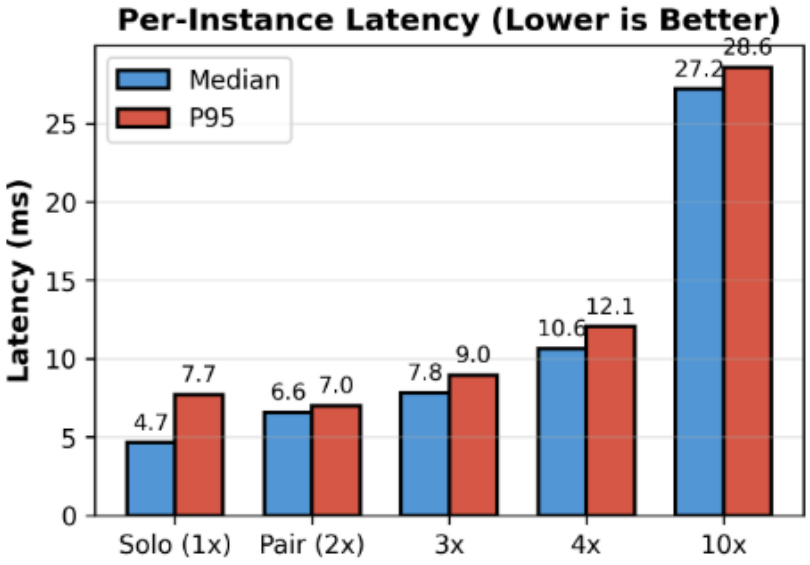

Figure 4: Average distilGPT2 latency as the number of models increases

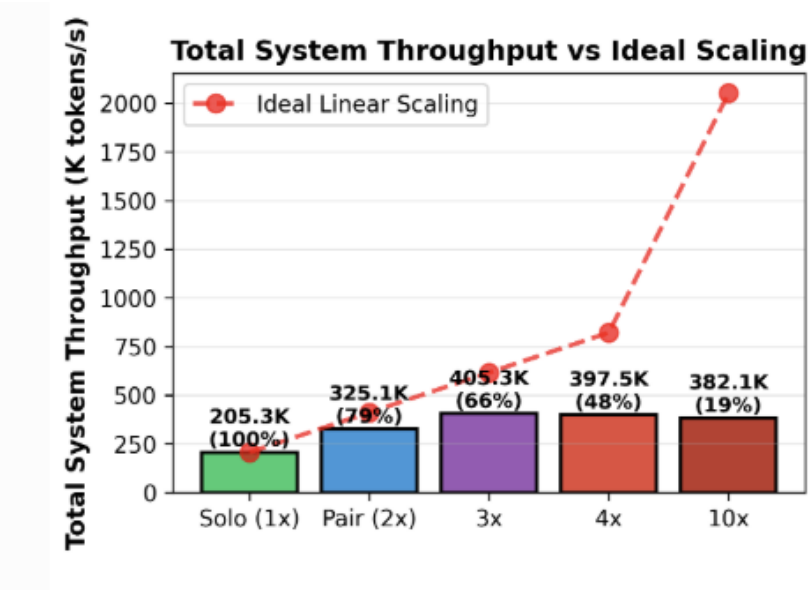

Figure 5: Average distilGPT2 throughput as the number of models increases

We also looked at how a scaling number of users affected performance. Naturally the latency increases since there will be some queue time waiting for the processes. However, the throughput can be seen going up. This is the goal of MPS, as it is able to overlap kernel execution and launches, and launch different processes' kernels together. This is reflected in by the Nsight data which shows us that that the mean compute warps in flight increases from 8% to 11%, to 13.7%, to 14.3%, and

14.8% for 1, 2, 3, 4, and 10 models respectively. Therefore, the main performance increase is that the scheduler is able to schedule multiple kernels as the number of clients increases. However, it is interesting that the increase in the number of warps scheduled drastically slows down after having 3 models running. This tells us that the number of execution units should be the bottleneck for inference time model execution.

We are speculating at this point, but it is likely that the models take a large amount of register space per thread, which results in each SMs register resource being full, but there are still more compute resources available. We are not sure how to fix this because we are not ML engineers, but it seems that either the hardware or model needs to be changed in order for all resources to be used to its maximum capacity. Having this as a bottleneck would make a lot of sense for why we also see very minimal pressure on the DRAM bandwidth.

Another interesting factor to note is that in the experiment with 10 distil GPT2 models, the spread for each models performance varies drastically from 23.64ms to 28.57 ms, which is a 20% difference in performance. All of the models and inputs are the same in this test case, so the runtime latencies are expected to be very similar. After examining the Nsight data, there does not seem to be any reason for the large disparity in run time latencies. It is possible that one model was luckier in terms of cache hits. It is also possible that the scheduler was biased in some sense, but this is unlikely. It can also be seen from the graph, that there was some ordering for model latency, which is likely due to the high number of processes and the small models. Since the resources are limited and all of the models are waiting for resources, there will inevitably be some ordering of the resources and who gets the resources first. If we were to have many larger models running together, the impact would be diminished since, receiving resources 1 time slice earlier would have a much smaller impact on the overall latency. Interestingly we have not seen a compounding effect of the overhead created by the additional processes. Normally with so many processes running at the same time, overhead would start to have considerable impact on CPUs.

Another part of our experiment was to see if there was any benefit to running encoders and decoders together, since one was expected to be more memory bound, while the other is compute bound. However, we found that this hypothesis did not hold, and in some cases it was detrimental to performance. For all of our experiments, the DRAM bandwidth was never pushed past 20% for reads and 10% for writes. This shows us that the models we chose were not necessarily memory bound in the way that we had expected. This could be due to the models we chose being too small to stress the GPU. Furthermore, in the experiment with 3 roBERTA models and 1 llama model, when the roBERTA models were running, the compute warps in flight was around 15.6%, but after they finished, the percentage went to 22.7% average, which is what the llama model achieved on its own. This goes to show that combining different architecture models does not necessarily mitigate multi tenancy effects.

It should also be noted that MPS did minimize the impacts of multi tenancy. When the smaller models are run with the larger models, we are able to see that the latency increase is not linear with the number of occupants, as shown in the case with 5 distil gpt2 models and 1 llama model. The models had a slow down varying between 244% and 279%. However, this could also be due to the fact that the smaller models are faster and have minimal impact on the compute resources

being taken. Examining the llama and mistral results we are able to see that llama only had a roughly 81% increase in latency, and mistral had an increase in latency around 78%. With the information from Nsight that the frequency and SM usage did not significantly change, we can attribute the minimized increase in latency to the MPS scheduler minimizing overhead between kernel launches and waiting for requests.

Assuming that we were restricted by the available register space we had, which is in the SMs, then the problem becomes compute bound. This means that no matter what combination of models we use, we will never have enough registers on the SMs to utilize the remaining warps and available memory bandwidth.

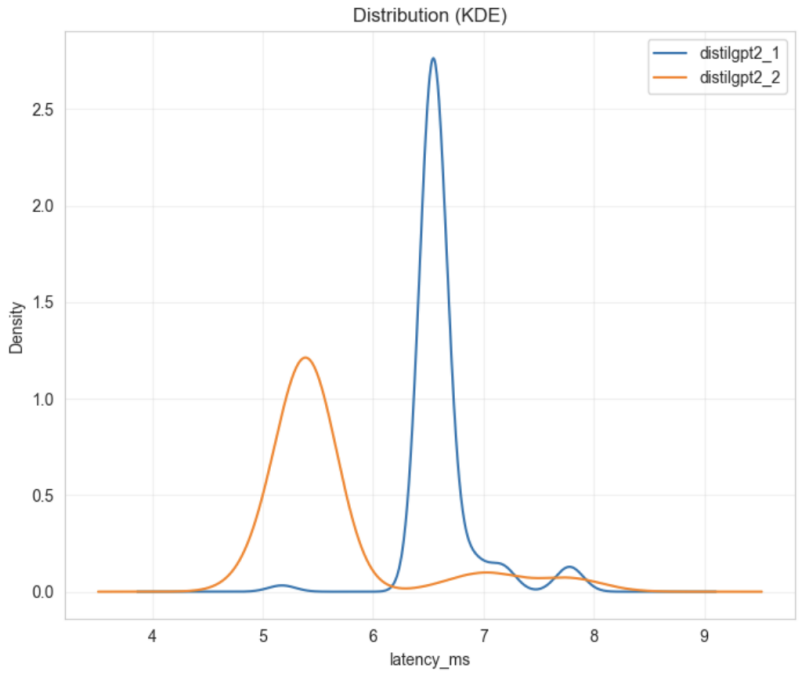

Figure 6: Pair distil GPT2 models Kernel Density Estimation Graph

We also ran the distil GPT2 pair without MPS to get a baseline of how MPS affects performance. The result is very clear that without MPS, a first come first serve policy is used.

The goal of our project was to find patterns in the resource usage between different multi tenancy paradigms for possible scheduler protocols. It is clear that

For future research, we would need to figure out a way to actually achieve memory bandwidth bottlenecks, so that we can actually stress test other parts of the GPU.

Other things to note is that there are other tools created by NVIDIA to partition the SMs as a percentage between different clients(processes). However, this is not dynamic in the sense that after it is set, it can not change, and if a client's SM is idle it can not be used by other clients.

This is an issue in the sense that it becomes difficult for QoS agreements to be held. Also, it is not possible to prioritize which clients get serviced by the MPS server.

Therefore a possible scheduler would be able to prioritize urgent kernels and also partition the SMs dynamically. This would mean allocating as many SMs as possible to balance which clients get more resources and if a client is dormant, shifting the SMs to other clients. One possible approach to be able to utilize all SMs while still dynamically allocating SMs, is to time slice kernels that are running on SMs reserved for other clients. This way if the other client suddenly needs compute resources, the scheduler can easily stop the kernels and clear the SMs. This would introduce more overhead, but more efficient usage of empty SMs while still allowing partitioning of resources.

Discussion and Conclusion

In this work, we conducted multi-tenant GPU experiments with GPU profiling to observe interference caused by multi-tenancy, identify the effects of this interference on a small variety of model classes, and look for configurations that allow for multi-tenancy and enhanced throughput with minimal latency increase.

In our experiments, we found the kernels to be compute bound rather than memory bound, although experiments were run with a relatively small batch size, and smaller models than are currently common in production systems. Regardless, we observed that smaller models experience disproportionate slowdown when cohabiting a GPU with larger models, which appear to dominate compute resources. However, it should be noted that under real workloads, requests to different models are unlikely to perfectly overlap, indicating the need for simulated conditions.

Overall, while we verify that multiple copies of the same model produces fair slowdowns, and that larger models monopolize compute resources, leading to possible starvation for smaller models, the current experiments are limited in scope. Some future directions to explore are larger batch sizes and sequence lengths, with the goal of saturating memory bandwidth, as well as experimenting with other model types. Finally, while MPS fairly schedules kernels of different models, we could experiment with size-aware scheduling, seeking to prioritize small workloads.

Figures

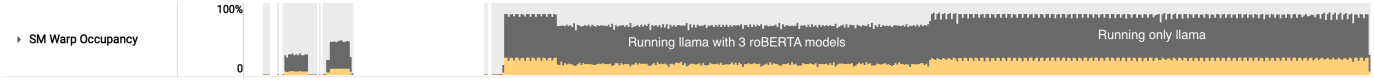

Figure 7: Nvidia Nsight output for 3 roBERTA and llama experiment

Credit Distribution

Akira: Developing experiment source code, config templates, data collection and profiling scripts, and misc utilities. Conducting data collection.

Tianyou: Model selection, experiment formulation, developing analysis tooling and notebooks, data analysis.

50%-50% work performed.

References

NVIDIA. *Multi-Process Service (MPS).* https://docs.nvidia.com/deploy/mps/index.html. Accessed: 2025-12-10.

Recasens, Pol G., Ferran Agullo, Yue Zhu, Chen Wang, Eun Kyung Lee, Olivier Tardieu, Jordi Torres, and Josep Ll. Berral. 2025. "Mind the Memory Gap: Unveiling GPU Bottlenecks in Large-Batch LLM Inference." *arXiv preprint arXiv:2503.08311,* https://arxiv.org/abs/2503.08311.