WattWiser: Power & Resource-Efficient Scheduling for Multi-Model Multi-GPU Inference Servers

Ali Jahanshahi
University of California, Riverside
Riverside, CA, USA
ajaha004@ucr.edu

Mohammadreza Rezvani
University of California, Riverside
Riverside, CA, USA
mrezv002@ucr.edu

Daniel Wong
University of California, Riverside
Riverside, CA, USA
danwong@ucr.edu

***Abstract*—With the increasing integration of Machine Learning (ML) applications into cloud services, providing high throughput Machine Learning inference serving has become a major demand for cloud service providers. The inference requests need to respond with bounded latency for each request to maintain a consistent Service-Level Objective (SLO). To ensure SLO, inference servers are equipped with multiple GPUs to satisfy the computational requirements. However, multi-GPU systems are extremely power-hungry. To resolve this, it is ideal to consolidate the load to a sub-set of GPUs, and potentially share GPUs, in order to minimize power consumption, without violating SLO.**

By consolidating GPUs and potentially sharing GPUs we can reduce the power consumption of multi-GPU inference servers. However, multiple inference models typically share the same inference server, which adds significant challenges in multi-model multi-GPU inference server environments. In this paper, we explore the challenges that this brings in achieving power efficiency. We introduce WATTWISER, a model management and scheduling policy that achieves power savings in multi-model environments where GPUs are shared. Our results show that WATTWISER can reduce power consumption by 34% while serving multiple models and maintaining the SLO.

***Index Terms*—Power efficiency, GPU, Inference Server**

I. INTRODUCTION

The demand for cloud-based inference solutions has prompted the creation of numerous APIs, frameworks, and hardware accelerators. Notably, Google's Cloud Inference platform has APIs for executing inference queries on large-scale typed time-series data, and NVIDIA's Triton Inference Server [1] is an open-source inference server that is optimized for NVIDIA GPUs. Similarly, Qualcomm Cloud AI 100 inference accelerator and Intel Nervana NNP-I (Spring Hill) have released hardware accelerators for cloud inference solutions.

While application-specific accelerators (ASICs), such as Google's Tensor Processing Unit (TPUs), offer lower power consumption and higher performance for cloud inference services, GPUs remain popular in data centers due to their programmability and support for general-purpose computing. The integration of more GPUs into cloud systems has led to power-hungry multi-GPU systems that present new power management challenges during design and deployment of data centers running machine learning-heavy workloads.

GPUs are designed for maximum efficiency at peak utilization, but this is not always the case in machine learning inference execution which tends to under-utilize the GPU [2]–[5]. To optimize GPU utilization, concurrent processing of inference requests is necessary. Moreover, the request-response nature of inference workloads varies throughout the day due to usage pattern fluctuations, leading to another source of potential under-utilization, which poses challenges to energy efficiency and can be exacerbated without proper coordination and management. [6]

To improve the efficiency of multi-GPU inference servers, prior works have explored how to spatially partition the GPU to improve utilization [7]. NVIDIA Multi-Instance GPU (MIG), as a hardware-supported feature, has been used to partition and isolate the GPU's resources to smaller slices to be used for inference [3]. NVIDIA Multi-Process Service (MPS), as a software/runtime feature, also have been used to allocate a provisioned percentage of the GPU resources to a process [4], [5]. However, both MIG and MPS allocated GPU resources statically which requires offline profiling of the optimal amount of resources required by the ML network [8]. More recently, fine-grain kernel-level spatial partitioning techniques [2] have been proposed to enable fine-grain kernel-scoped spatial partitioning, but this still requires off-line profiling of individual kernels. While all of these works improved GPU utilization, they do not necessarily target power efficiency directly nor address the unique challenges that multi-model multi-GPU inference servers present.

GPU-NEST [9] is the most relevant work to directly investigate the power characteristics of multi-GPU inference servers. GPU-NEST has shown different sources (CPU-GPU communication, GPU hardware resources, and scheduling) can impact the energy efficiency and QoS of multi-GPU inference servers among which scheduling policy plays an important role in the overall energy- and resource efficiency of the inference system. However, that work was limited to only exploring inference servers that run a single inference model at a time.

In this paper, we propose WATTWISER, a scheduler to address the power-efficiency issues of *multi-model* multi-GPU inference servers. Our contributions are as follows:
• In Sec. II, we identify challenges and opportunities for power savings in modern multi-model multi-GPU inference servers.
• In Sec. III, we introduce WATTWISER, our model management framework to enable power-efficient inference. WATTWISER dynamically determines the number of active GPUs and how to schedule incoming requests across allocated GPUs to minimize power during varying ML inference loads.

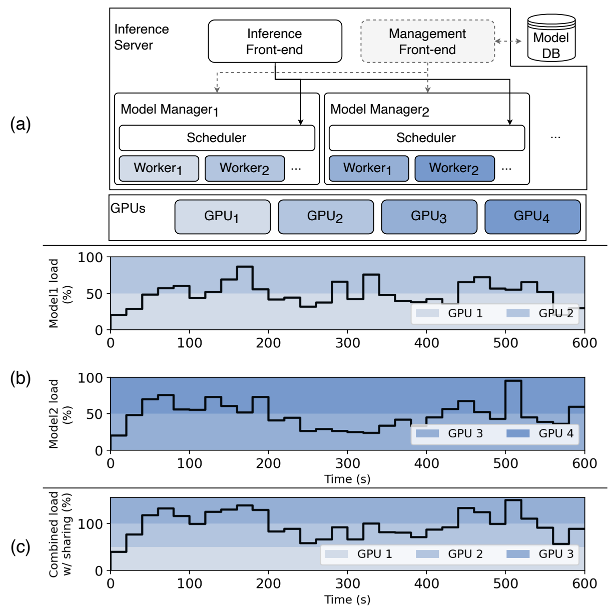

Fig. 1: a) High-level overview of multi-GPU inference server design. Model manager instantiates one worker per GPU assigned to the model. b) $Model_1$ and $Model_2$ workload traces over time normalized to maximum throughput of 2 GPUs for each while **sharing no GPUs** on the server. c) Combined load of $Model_1$ and $Model_2$ **with GPU 2 shared**.

• In Sec. IV, we thoroughly evaluate WATTWISER and demonstrate the efficacy in achieving power savings under multi-model multi-GPU inference server systems.
• We show that WATTWISER is able to reduce average power by 34% compared to the baseline scenario.

II. Background and Motivation

Multi-model multi-GPU inference servers. Figure 1(a) shows the architecture of a *multi-GPU inference server*. For each model that is loaded, there is a *model manager* that handles the inference. Each model manager consists of a *scheduler* that schedules incoming inference requests to a set of *workers*. Each worker runs an *inference backend* (such as pytorch, Tensorflow, etc.) and performs the inference requests using a GPU that it is paired with. It's possible for multiple workers to *share* a single GPU, even across two model managers (as shown in Figure 1(c)).

When a model is loaded, the inference servers allocate a dedicated set of GPUs to each model in order to facilitate inference requests for multiple models simultaneously. For example, each ML model is allocated GPU 1,2 and 3,4, respectively, in Figure 1(b) and each model is allocated GPU 1,2 and 2,3, respecitively, in Figure 1(c). The number of GPUs assigned to each model is determined by the application's required maximum throughput, measured by requests per second (RPS).

Under-utilization in multi-GPU inference servers. The variability in inference workloads due to usage pattern fluctuations can result in under-utilization of GPUs. For example, Figure 1(b) shows two workload traces[1]. In this illustrative scenario, each model is allocated 2 GPUs and the load shown is normalized for the maximum load that 2 GPUs can support (100% meaning 2 GPUs are fully utilized by the model).

As the total load in Figure 1(b) fluctuates, there are periods of time where the allocated GPUs are underutilized and does not require all 4 GPUs, but can fit into 3 GPUs if the GPUs are shared. Figure 1(c) depicts this scenario where each model are allocated with a shared GPU and shows how the same load can be handled by 3 GPUs with higher utilization.

This presents an opportunity for power savings by consolidating model inference to a subset of GPUs. However, there are many challenges towards consolidating models as it would require sharing a GPU between multiple inference models, potentially leading to interference and tail latency violations. *Therefore, consolidating model inference into a subset of GPUs needs careful coordination.*

Related Work. Most prior works on multi-model inference servers have focused on improving the utilization of GPUs through spatial partitioning techniques. For example, PARIS and ELSA [3] and GPUlet [4] explored how to allocate models across multi-GPUs through spatial partitioning the GPUs through Nvidia's MIG and MPS, respectively. GSLICE [5] and KRISP [2] explored how to spatially partition and share a single GPU to maximize the number of models that can run on a GPU using MPS and a novel kernel-scoped spatial partitioning technique, respectively.

The most relevant work is GPU-Nest [9], which showed the impact of inference request scheduling policy on energy-efficiency of multi-GPU servers. It showed that the baseline scheduling policy in Nvidia's Triton inference server is not energy-proportional due to uniformly scheduling requests equally across all GPUs. It also demonstrated that consolidating GPUs may lead to power savings. However, this work does not explore how to actively manage models and GPUs when multiple models are handled by the inference server.

Summary. Clearly, there exists a gap in work that explores the power efficiency of multi-model multi-GPU inference servers. In order to improve the power efficiency of multi-GPU inference servers, we need to both improve the utilization of the GPUs and improve the scheduling policies that handle multi-model GPU inference. In this work, we first show the effect of scheduling policy on GPU utilization and power-efficiency in multi-GPU multi-model inference servers. Then, we propose a scheduling policy to enable consolidating GPUs among models while maintaining SLO.

III. WattWiser

In this section, we present WATTWISER, a power-efficient model management framework for multi-model multi-GPU inference servers. WATTWISER dynamically determines how many GPUs should be active and manages the loading/unloading of workers. Next, WATTWISER determines how to

[1]Traces derived from Facebook [10]

distribute and schedule incoming requests across the available GPUs in order to minimize power consumption while satisfying SLO targets. We will first highlight the scheduling challenges in a multi-model multi-GPU inference server. Then we will detail how WATTWISER's scheduling policies enhance power-efficiency multi-GPU inference servers.

A. Dynamic GPU allocation/deallocation

In order to save power, WATTWISER aims to enable a minimal subset of GPUs that can support the total load of the system for serving inference requests. In WATTWISER, the inference server monitors the incoming RPS rate of each model. Based on the incoming RPS rate, it determines how many GPUs are required for that RPS rate (i.e. serving GPUs) and turns off under-utilized GPUs initially allocated to that model. To avoid thrashing the GPU by turning them on and off during run-time (due to incoming RPS rate variations), we use a hysteresis to determine when to turn a GPU on/off.

This dynamic GPU allocation/deallocation policy is formalized in Algorithm 1, `AllocateGPUs()`. At runtime, server monitors the current load of each model (`current_rps`) at 1-second intervals. In order to detect if more GPUs are needed for the current load (RPS) of the model, in `more_GPUs_needed()`, `current_rps` is compared to *model_max_rps* as the threshold. This threshold is used to avoid SLO failure in case of a surge in the load (RPS). If `current_rps` decreases to the point where the *serving set* of GPUs are underutilized, in `less_GPUs_needed()`, we remove a GPU from the *serving set*.

At a high-level, we are essentially "packing" the total workload into a sub-set of available GPUs. This is similar to the packing policy proposed in GPU-NEST [9] which identified that by packing workloads into a sub-set of available GPUs, you can realize power savings by turning off unnecessary GPUs. However, as we will see later, GPU-NEST can only handle inference servers that run a single inference model. *We will now demonstrate how prior works cannot handle inference servers that manage multiple models at the same time.*

B. Scheduling policies

Once WATTWISER determines the number of GPUs that should be utilized for the current load, the next decision is how to schedule the incoming requests across the GPUs. Scheduling has a large impact on power efficiency and tail latency in multi-modal inference server environments. We will first highlight the impact of scheduling and then propose several scheduling policies.

Algorithm .1: Allocation policy

```
cur_rps: current RPS.
model_max_rps: model's Max RPS a GPU can support.
allocated_gpus: GPUs initially allocated to the model.
serving_set: GPUs that serve requests

AllocateGPUs():
|  if more_GPUs_needed(cur_rps, model_max_rps)
|  |  AddGPU(allocated_gpus, serving_set)
|  else if less_GPUs_needed(cur_rps, model_max_rps)
|  |  RemoveGPU(serving_set)
|  UpdateSharingWeights(); or
|  UpdateSharingLoadWeights();
```

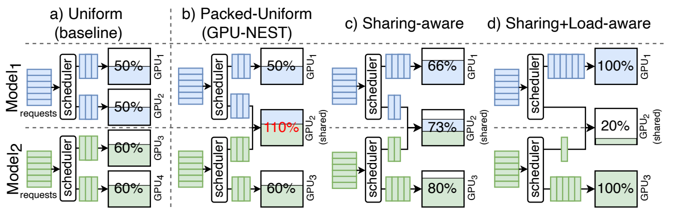

Fig. 2: GPUs utilization with different scheduling policies for two models. Uniform (a,b) and Sharing-aware (c) both schedule requests to the shared GPU regardless of the load. Sharing+Load-aware (d) policy utilizes the maximum capacity of non-shared GPUs before scheduling requests to the shared GPU.

1) Impact of scheduling in multi-model multi-GPU inference servers : As mentioned in the previous section, workloads are mostly operated below their maximum load which creates an opportunity for power saving through resource (GPU) sharing. As an example, Figure 2(a) illustrates an example of the uniform scheduling effect on GPU utilization in a scenario where no GPU is shared among two models. Each model is allocated two GPUs with the requests *uniformly* scheduled across all GPUs. This is the baseline behavior of existing inference servers, such as Nvidia's Triton inference server [1].

However, the dynamic load of the server allows us to fit the load within 3 GPUs if we can share a GPU between both models. Figure 2(b), shows a scenario where one GPU is shared between 2 models leading to utilizing 3 GPUs in total. This scenario shows GPU-NEST's scheduling policy where the schedulers in each model manager are not coordinated and schedule requests uniformly across their allocated GPUs without knowledge that the backend GPUs are shared. This can lead to over-utilization of the shared GPU (i.e. GPU_2) and tail latency violations. Therefore, to satisfy tail latency requirements, the *schedulers from different model managers should be coordinated* or be made aware of backend GPUs that are shared and the load distribution across GPUs.

2) Sharing-aware Scheduling: To support multi-model inference, we propose two scheduling policies: a *Sharing-aware scheduling* policy and a *Sharing+Load-aware scheduling* policy. Both policies take into account the GPUs that are shared when packing multiple inference models into a subset of active GPUs while maximizing energy efficiency.

We first present a *Sharing-aware Scheduling* policy where the model manager's schedulers are made aware of how many models are sharing a particular GPU. In Sharing-aware scheduling, requests are scheduled to each GPU *proportional to the number of models sharing the GPU*. For example, in Figure 2(c), each inference model is allocated to two GPUs (`allocated_GPUs = 2`), with the middle GPU (GPU_2) being shared. Therefore, GPU_1 has a sharing factor of 1, and GPU_2 has a sharing factor of 2. $Model_1$'s scheduler interleaves the requests among GPU_1 and GPU_2

Algorithm .2: Scheduling policies

```
cur_rps: current RPS.
model_max_rps: model's Max RPS a GPU can support.
sharing_factor: number of models sharing each GPU.
gpu_rps: RPS load mapped to each GPU.
weights: request distribution weights of each GPU.

UpdateSharingWeights():
|  remained_rps = cur_rps
|  // Calculate weights w.r.t GPU's sharing_factor
|  lcm = LCM(sharing_factor)
|  for gpu in allocated_gpus
|  |  weight[gpu] = lcm/sharing_factor[gpu]

UpdateSharingLoadWeights():
|  remained_rps = cur_rps
|  // Prioritize scheduling load to non-shared GPUs
|  for gpu in not_share_gpus
|  |  gpu_rps[gpu] = model_max_rps
|  |  remained_rps -= gpu_rps[gpu]
|  // Schedule the remaining load to shared GPUs
|  for gpu in shared_gpus
|  |  shared_rps = model_max_rps/sharing_factor[gpu]
|  |  if remained_rps < shared_rps
|  |  |  gpu_rps[gpu] = remained_rps; break
|  |  else
|  |  |  gpu_rps[gpu] = shared_rps
|  |  remained_rps -= gpu_rps[gpu]
|  // Calculate weights for request distribution
|  min_gpus_rps = min(gpu_rps)
|  for gpu in serving_set
|  |  weight[gpu] = gpu_rps[gpu]/min_gpus_rps
```

with the ratio of 2:1. By interleaving requests among GPUs based on their *sharing ratio*, it aims to avoid contention on the models that share GPUs but yet maintain a load balance between the allocated GPUs. This policy is shown in Algorithm 2 `UpdateSharingWeights`. Based on the `sharing_factor`, we can compute the `weight` assigned to each GPU by dividing the least common multiple (LCM) of the `sharing_factor` by each GPU's `sharing_factor`. During runtime, the requests are scheduled to the GPUs using a weighted round robin scheduling where every GPU gets `weight` requests (2 requests for GPU_1, 1 request for GPU_2 in Figure. 2). In our inference server, the Management front-end, upon user's request for loading a model into a GPU, broadcasts the `sharing_factor` of each GPU to all model's scheduler to make them aware of the sharing status of each GPU.

In Sharing-aware scheduling, even though the contention is reduced while GPUs are shared, it does not take the current load of the model into account. For example, in scenarios where the load is too low, Sharing-aware scheduling still schedules inference batches to the shared GPUs creating unnecessary contention as well as under-utilization of the non-shared GPU. To avoid contention and under-utilization, we propose a Sharing+Load-aware scheduling policy.

3) Sharing+Load-aware Scheduling: We now present *Sharing+Load-aware scheduling*, which aims to minimize contention between different inference models while improving utilization of GPUs, which is the most energy efficient operating point. Figure 2(d) illustrate an example of how the Sharing+Load-aware scheduler distributes the requests to GPUs. Sharing+Load-aware prioritizes the GPUs that are *not shared* and can support inference requests for the current load without violating SLO. This maximizes the utilization of non-shared GPUs and minimizes the contention in shared GPUs. For example, $Model_1$ and $Model_2$ prioritize and maximize the utilization of the non-shared GPU_1 and GPU_3, respectively. Then only when necessary, do the requests spill over to the shared GPU, GPU_2.

In Algorithm 2, `UpdateSharingLoadWeights()` shows the pseudo-code of the Sharing+Load-aware scheduling policy. The policy first distributes `model_max_rps` load to the non-shared GPUs and any remaining load is spilled over to the shared GPUs. We then normalize the load to each GPU (`gpu_rps`) to get a `weight` ratio for each GPU. For example, in Figure 2.d, $Model_2$ will have a `weight` of 5 and 1 for GPU_3 and GPU_2, respectively. During runtime, the requests are scheduled to the GPUs using a weighted round robin scheduling where every GPU gets `weight` requests (5 requests for GPU_3, 1 request for GPU_2, etc.).

IV. Evaluation

A. Evaluation Methodology

Server Hardware: We deployed our inference server on a system with 4 AMD MI50 GPU, 2 AMD EPYC 7302 16-Core Processor, 512 GB RAM, Ubuntu 18.04 LTS with kernel 5.4.0. The system runs the AMD ROCm 5.2 runtime stack.

Workloads: We use a combination of computer vision pre-trained models for evaluation. Table I shows the model as well as their SLO and max throughput. For each model, SLO is measured similarly to prior works where we set 2x the isolated inference tail latency [2]–[4]. WattWiser's framework support serving any number of models on servers with any number of GPUs. To illustrate the efficiency of WattWiser, we deploy 2 models on 4 GPUs (2 GPUs allocated for each model) as the workload. Our workload consists of two models (`resnext101` and `resnet152`) with high latency, one model (`vgg19`) with medium latency, and two models (`alexnet` and `squeeznet1`) with low latency requirements. To evaluate our approach, we picked five combinations of workloads with different latency requirements: `resnext101-resnet152` (high-high), `resnet152-vgg19` (high-medium), `resnext101-squeezenet1` (high-low), `vgg19-alexnet` (medium-low), and `alexnet-squeezenet1` (low-low) pairs.

Workload traces: For workload traces, we use Facebook SWIM traces [10] shown in Figure 1(b). Each trace is nor-

TABLE I: Inference workload used with their maximum throughput on single GPU. SLO is 95% tail latency of the model on one GPU.

Model	Max Throughput (RPS)	SLO (ms)	Latency
`resnext101`	140	101	High
`resnet152`	150	76	High
`vgg19`	320	41	Medium
`alexnet`	1750	25	Low
`squeezenet1`	2050	28	Low

malized to [20%-95%] of our system. For each model, 100% load has been mapped to the maximum throughput (RPS) two GPUs can support for that model without violating the SLO.

Multi-GPU Inference server: We created our own custom inference server framework as most existing inference servers, such as Triton Inference Server [1], are designed for Nvidia-based GPU systems and tightly integrate Nvidia-specific features. Our multi-GPU inference server reflects the design of Nvidia's Triton and is shown in Figure 1(a). Our multi-GPU inference server consists of the following components. *Inference Front-end*: a thread responsible for accepting concurrent requests from clients, routing them to the requested model's *Scheduler*, and sending back the inference result (response). *Management Front-end*: a thread that provides an API to load models from the Model Database for serving, controlling the number of GPUs, number of workers, and batch size for each model. *Model Database*: containing all of the pre-trained models. *Model Manager*: there is one model manager per model containing a *Scheduler* thread (for batching and distributing requests to workers), and one or more workers per GPU that is allocated to the model. *Workers*: a concurrent thread that performs pre-processing, inference, and post-processing on a batch of requests. Each worker is independent of one another and uses an independent stream, allowing for concurrent inference execution on the same or different GPUs. *Communication*: all components of the server communicate through highly scalable and efficient event-based *ZeroMQ* library [11]. *ML Backend*: We use PyTorch 1.12 as the worker ML backend.

Power measurement: To measure the power consumption of the server we used AMD's ROCm system management API (*rocm-smi*). The samples of power are taken at 10ms intervals for each experiment and are averaged for each trace point in our evaluations.

B. Evaluation results

WattWiser in action: in WATTWISER, loading and unloading models into the server is performed upon user request through *Management front-end*. Users are able to request to load a model to a given GPU. Upon loading/unloading each model to/from the GPUs, WATTWISER automatically updates the `allocated_GPUs` set and `sharing_factor`, and adjusts resources (GPUs) based on the selected scheduling policy while serving the requests. Figure 3 illustrates the workings of WATTWISER while two inference models are following variable request traces. As the load fluctuates, WATTWISER dynamically consolidates the total load into a subset of active GPUs as shown in Figure 3(c). In the figure, each bar at each time step corresponds to the percent of requests scheduled to each GPU (labeled on right axis). For example, in time step 1, both model's load is below 50% (❶) which means each model only needs one GPU to serve requests for the model (100% load means 2 GPUs are required). Therefore, all (i.e. 100%) of the requests received during time step 1 are scheduled to each model's dedicated GPUs leaving shared GPU (GPU 2) free to be turned off. In

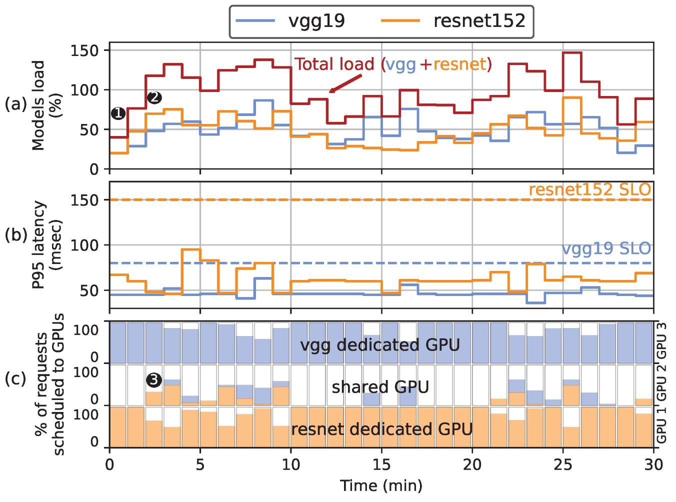

Fig. 3: Timeline view of WATTWISER in action. WATTWISER is able to dynamically adapt the number of serving GPUs allocated to each model based on their sharing status (Sharing-aware) as well as the rate of scheduling requests to each GPU based on load (Load-aware) (c)[2] while maintaining tail latency (b) under variable loads (a).

time step 3, `resnet152` load goes above 50% (❷) therefore, some of the requests (based on policy explained in Section III) are scheduled to the shared GPU (i.e. GPU 2, ❸).

WATTWISER is able to maintain the inference model's tail latency (solid lines in Figure 3(b)) by staying within the SLO's target tail latency (dashed lines in Figure 3(b)). Overall, this allows WATTWISER to obtain power savings by turning off the unutilized shared GPUs (rectangles that are completely white in Figure 3(c)) under variable load in a multi-model multi-GPU inference server environment.

Tail latency: We first evaluate the impact of WATTWISER scheduling policy on tail latency. We used request traces shown in Figure 1(b) for each of the five pairs of inference workloads as detailed previously. For each trace, a client was used to send requests to each pair of the models concurrently, following the load of the traces at each step for each model. The load specified at each step of traces is mapped to the max throughput of 2-GPUs (i.e. 2x the maximum throughput listed in Table I) for each model. For example, for `resnet152-vgg19` pair, we used the top and middle trace of Figure 1(b), respectively. While sending the requests, the client keeps track of the latency of all of the requests, and at the end of each pair's experiment, the 95th percentile latency is calculated and reported.

Figure 4 shows the distribution of normalized 95th tail latency for each model pair trace. The distribution illustrated in Figure 4 are the tail latencies for each step/load of the trace. We present the distribution of tail latencies rather than the overall tail latency of the whole trace to demonstrate a more complete overview of the behavior of different scheduling

[2]Each rectangle corresponds to the percent of requests scheduled to each GPU for each model at any step. The rectangles' height shows the percentage of requests scheduled to each GPU and *does not* represent GPU utilization.

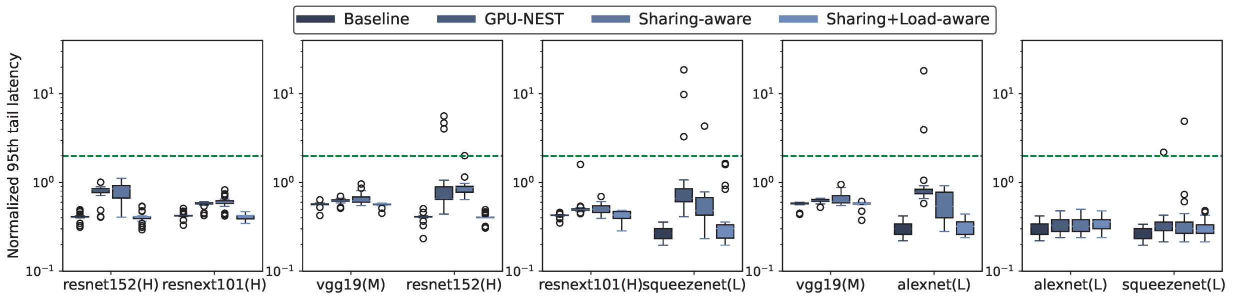

Fig. 4: Distribution of tail latencies for different model pairs at each load step of the inference model's traces in Figure 1.b. Green dashed line indicates target tail latency.

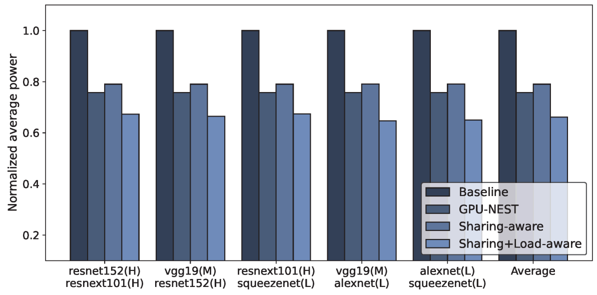

Fig. 5: Average power of each model pair while following the inference model's load traces.

policies on the tail latency. The horizontal green dashed line indicates the SLO's target tail latency.

Across all scheduling policies, the smaller model with lower latency's tail latency is more sensitive and more impacted when paired with larger models with higher latency.

Sharing+Load-aware policy across all model pairs has not failed the SLO of the model. The elevation in maximum normalized tail latency in Sharing+Load-aware, compared to baseline, is expected since we are sharing one GPU between two models while baseline has two dedicated GPUs for each of the two co-running models. Although the maximum normalized tail latency (the outliers) has been elevated compared to the Baseline (Uniform), it achieves better tail latency when compared to GPU-NEST (Packed-Uniform), and the Sharing-aware policy. This is due to Sharing+Load-aware's aim to reduce contention in the shared GPU.

In some cases, for example, the `resnet152-resnext101` pair, Sharing-aware performs worse than GPU-NEST. Since the Sharing-aware schedules the requests to the non-shared and shared GPUs with a ratio of 2:1, it may cause over-utilization of the non-shared GPU in higher loads leading to the growth of the host-side queue and impacting the tail latency. GPU-NEST (packed-uniform), however, in higher loads, distributes the requests among the non-shared and shared GPUs evenly and since the total load of the model can be supported by two GPUs in higher loads, it performs better. This issue does not exist in Sharing+Load-aware policy since the requests are distributed among non-shared and shared GPUs by adjusting the GPUs' `weight` based on the load. This demonstrates the need to be aware of both load and which GPUs are shared.

Power savings: To investigate the impact of scheduling policies on power consumption, we measure the GPU's power for each model pair while following the load traces. Figure 5 shows the average power consumption of GPUs utilized by each model in each experiment normalized to the Baseline Uniform case. By consolidating the total load into a subset of active GPUs, the average power consumption of the GPUs is reduced in all scenarios. On average, compared to baseline, Packed-Uniform saves 24%, Sharing-aware saves 20%, and Sharing+Load-aware saves 34% average power.

Sharing-aware scheduling policy, however, consumes more power compared to GPU-NEST's Packed-Uniform scheduling. As mentioned before, this is due to the fact that Sharing-aware policy distributes requests between non-shared and share GPUs with a ratio of 2:1 (compared to Packed-Uniform where the ratio is 1:1). Distributing requests with ratio of 2:1 utilizes the non-shared GPU, more leading to higher power consumption compare to Uniform. However, note from Figure 4 that packed uniform tends to violate SLO in almost all scenarios, except for `resnet152-resnext101`. So most of *GPU-NEST's power savings gains come at the cost of increased SLO violations.*

The Sharing+Load-aware policy achieves the best power savings across all workload mixes. This policy packs the requests to the non-shared GPU which increases the utilization (and energy efficiency) of the non-shared GPU. This also achieves the best tail latency as the amount of contention is minimized in the shared GPU.

V. Conclusion

Multi-model multi-GPU inference servers experience unique challenges in achieving energy efficiency. In this work, we highlight the challenges and limitations of existing multi-GPU inference server scheduling policies. We present WattWiser, a framework to manage GPUs allocation and energy-efficient scheduling policies for multi-model multi-GPU inference servers. We show that WattWiser can save power by 34% on average while maintaining target tail latency.

Acknowledgement

This work is partly supported by National Science Foundation under grants CNS-1955650 and CNS-2047521. We would also like to thank the anonymous reviewers for their invaluable comments and suggestions.

References

[1] Nvidia, “Triton Inference Server,” https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html.

[2] M. Chow, A. Jahanshahi, and D. Wong, “KRISP: Enabling kernel-wise right-sizing for spatial partitioned gpu inference servers,” in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023.

[3] Y. Kim, Y. Choi, and M. Rhu, “PARIS and ELSA an elastic scheduling algorithm for reconfigurable multi-gpu inference servers,” in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 607–612.

[4] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, and J. Huh, “Multi-model machine learning inference serving with gpu spatial partitioning,” *arXiv preprint arXiv:2109.01611*, 2021.

[5] A. Dhakal, S. G. Kulkarni, and K. Ramakrishnan, “Gslice: controlled spatial sharing of gpus for a scalable inference platform,” in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 492–506.

[6] J. Kosaian, A. Phanishayee, M. Philipose, D. Dey, and R. Vinayak, “Boosting the throughput and accelerator utilization of specialized cnn inference beyond increasing batch size,” in *Proceedings of the 38th International Conference on Machine Learning*, 2021.

[7] F. Yu, D. Wang, L. Shangguan, M. Zhang, C. Liu, and X. Chen, “A survey of multi-tenant deep learning inference on gpu,” 2022.

[8] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, and J. Huh, “Serving heterogeneous machine learning models on Multi-GPU servers with Spatio-Temporal sharing,” in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 199–216.

[9] A. Jahanshahi, H. Z. Sabzi, C. Lau, and D. Wong, “GPU-NEST: Characterizing energy efficiency of multi-gpu inference servers,” *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 139–142, 2020.

[10] SWIMProjectUCB, “Swim project,” 2013. [Online]. Available: https://github.com/SWIMProjectUCB/SWIM/wiki

[11] P. Hintjens, *ZeroMQ: messaging for many applications*. ” O’Reilly Media, Inc.”, 2013.